**实验一：系统软件启动过程**

**一、实验目的**

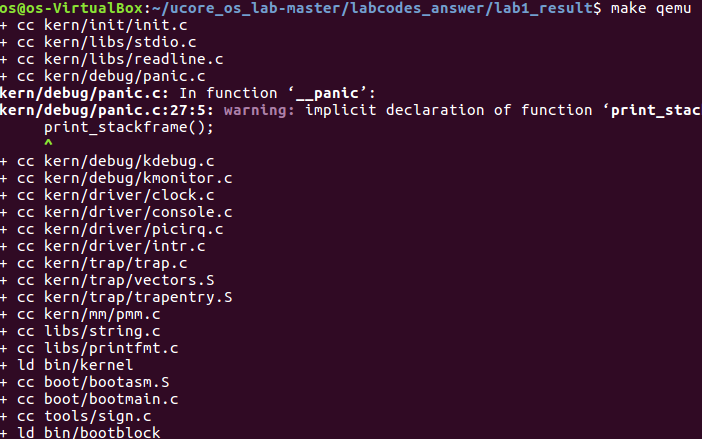
操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-bootloader来完成这些工作。为此，我们需要完成一个能够切换到x86的保护模式并显示字符的bootloader，为启动操作系统ucore做准备。

**二、实验内容**

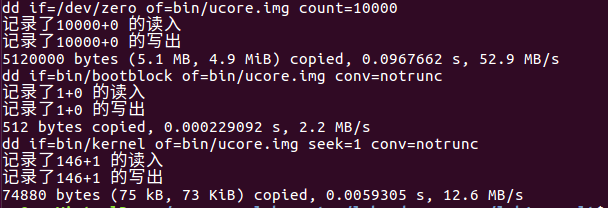
**1.理解通过make生成执行文件的过程。**

**操作系统镜像文件ucore.img是如何一步一步生成的？**

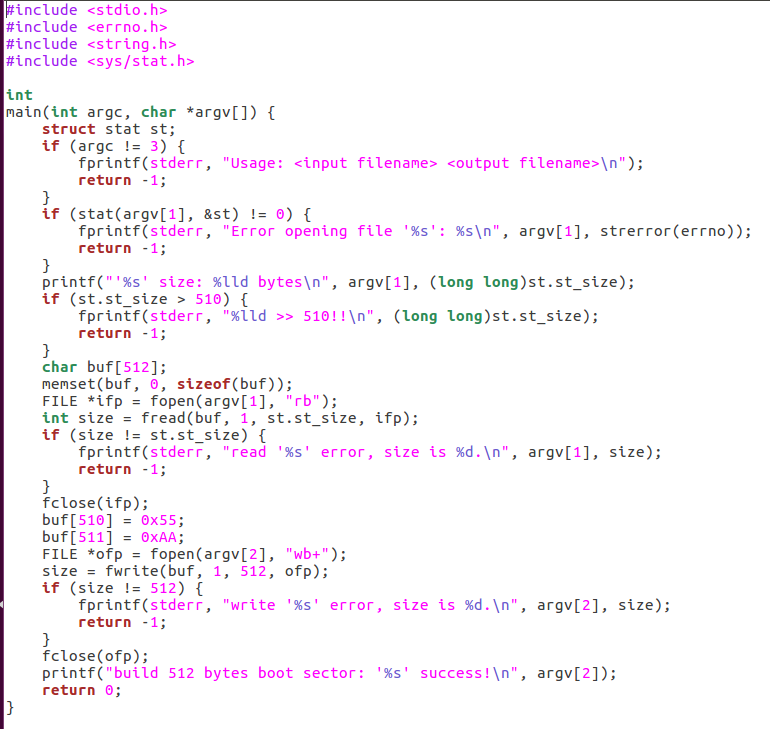
电脑首先针对kern文件下的.c和.S文件进行了编译，然后通过ld把kern文档下的所有程序和bin文件连接起来，并重定位他们的数据，然后生成kernel程序。其次电脑针对boot文件夹下面的.c和.s文件进行了编译，并且将tools文件夹下面的sign.c也进行了编译，然后通过ld生成bootblock程序。其次电脑针对boot文件夹下面的.c和.S文件进行了编译，并且将tools文件夹下面的sign.c也进行了编译，然后通过ld生成bootblock程序。因此我们可以了解到在生成ucore.img之前，需要先通过编译链接，生成kernel和bootblock的ELF文件。



接下来我们可以看到通过dd指令，将上面我们生成的kernel和bootblock的ELF文件拷贝到ucore.img当中，根据拷贝的顺序我们可以看到，首先是将bootblock拷贝进了ucore.img，然后才是将kernel拷贝进ucore.img，所以可以得出bootblock是引导区，kernel是操作系统内核。除此之外，在上文我们看到电脑还对sign.c进行了编译，然而在接下来的链接中并没有对其进行进一步操作，所以我们打开sign.c文件进行查看。



在这里我们看到给sign.c传进去了3个参数，其中通过输出，我们可以了解到argv[1]传入的是obj/bootblock.out：



紧接着在下面定义了一个长度为512的字符串，然后从obj/bootblock.out中读取500个字符存到buf里面，接下来在buf的结尾加上0x55AA，并且将其写到argv[2]里面，由于在运行中看不出来argv[2]是哪里，所以针对sign.c文件进行了修改，输出了argv[2]：







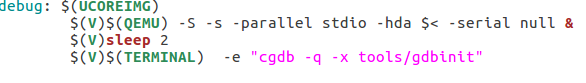
可以看到我们将更改后的buf存入了bin/bootblock当中，即我们的sign.c文件，在引导区内做了个0x55AA的标记，然后将其存入了bin/bootblock当中，然后拷贝到ucore.img里面。

自此，我们的ucore.img文件正式生成。

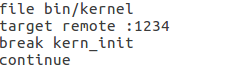
**2.使用qemu执行并调试lab1中的软件。**

(1) 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

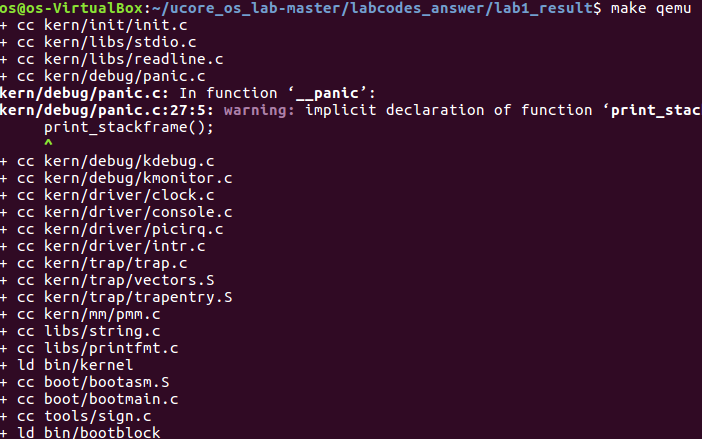
首先通过Makefile文件来查看make debug的相关操作：



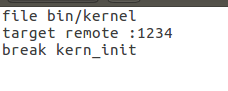
通过debug的定义，我们可以看到，在debug里首先是对qemu进行的操作，然后等待一段时间之后，再进行针对gdbinit文件进行的调试，所以，我们继续观察gdbinit文件：



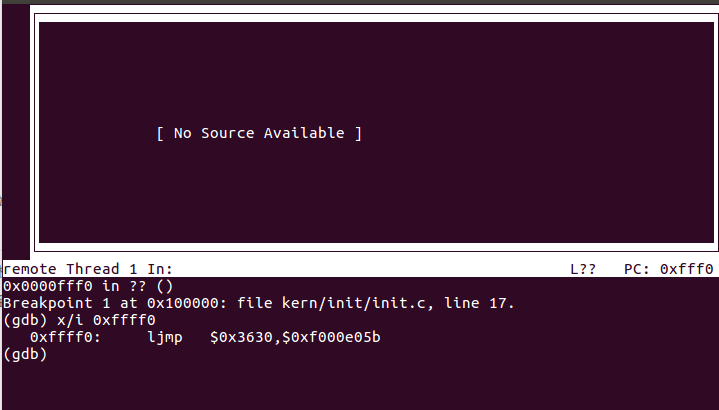
在gdbinit文件里我们可以看到电脑在运行到kern\_init是会触发break，然后又紧接着在下一步continue，所以会继续执行，再根据第一问的编译顺序：



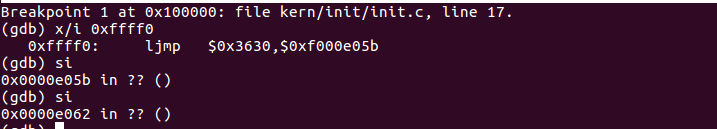
我们可以看到电脑CPU在加电之后第一步要执行的就是对kern\_init进行编译，同时这也是BIOS的第一条指令，所以我们需要在这里break，然后通过gdb一步一步针对BIOS进行运行，因此，我们将gdbinit中的continue删除，从而使得电脑停在这里。



然后进行make debug：



pc这个寄存器，所谓的pc值是通过CS:IP而得到的，因此这里的PC所代表的是eip寄存器里面的值。

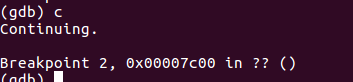


(2)在初始化位置0x7c00设置实地址断点，测试断点正常。

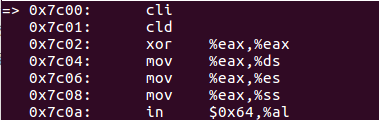
由于我们需要观察电脑在0x7c00处电脑的运行情况，所以首先需要在地址为0x7c00的地方设置断点，使得电脑在此停住：



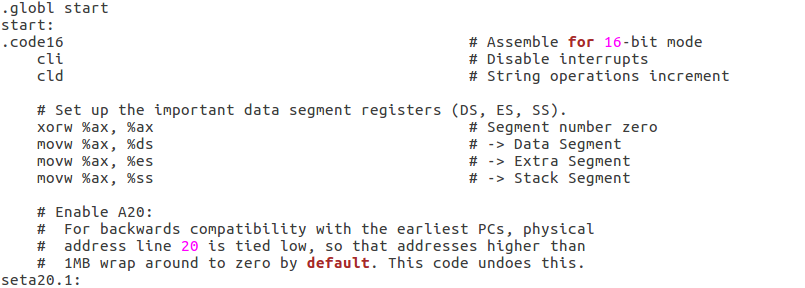
然后通过continue指令，使得程序继续运行，直到运行到0x7c00之后再次停住。



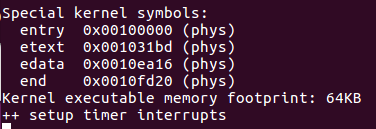
再次通过x/10i $pc查看相近的指令：



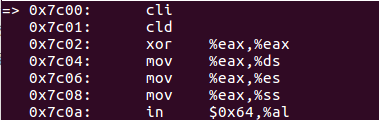
通过查看bootasm.S文件我们可以看到，此时电脑已经进入bootasm.S文件，开始执行相应的代码

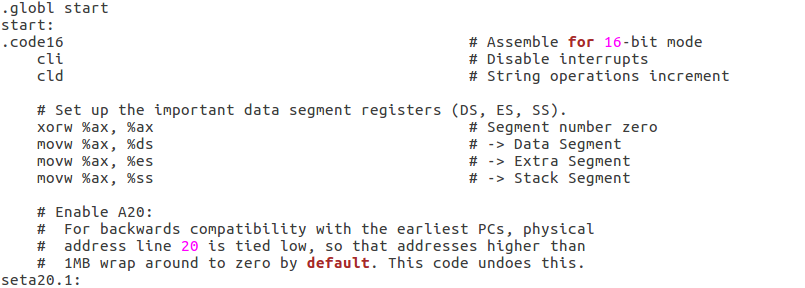


然后再次输入c我们看到qemu工作正常，所以断点正常。



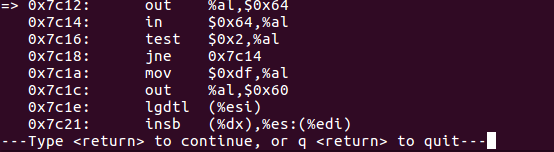
(3)从0x7c00开始跟踪代码运行，将单步跟踪反汇编得到的代码与bootasm.S和bootblock.asm进行比较。

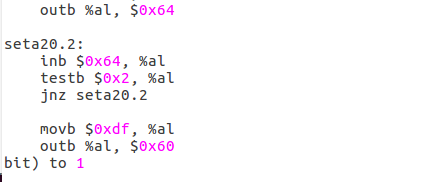




通过比较我们可以发现bootasm.S和bootblock.asm的汇编代码几乎相同。

(4) 自己找一个bootloader或内核中的代码的位置，设置断点并进行测试。我们将断点设在0x7c12，来进行测试：



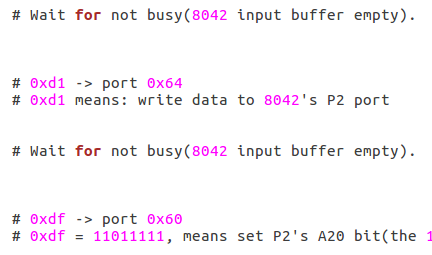


过对比我们可以看到，其汇编代码相同，然后输入c, qemu继续正常工作，所以断点正常。

**3.分析bootloader进入保护模式的过程。**

根据说明书我们可以知道，A20地址线由键盘控制器8042进行控制，我们的A20所对应的是8042里面的P21引脚，所以问题就变成了我们需要将P21引脚置1。

对于8042芯片来说，有两个端口地址60h和64h。对于这两个端口来说，0x64用来发送一个键盘控制命令，0x60用来传递参数，所以将P21引脚置1的操作就变成了，我们首先利用0x64输入一个写入的指令，然后由0x60读进去相应的参数来将P21置1。



在这里我们还需要注意一个问题就是当前端口(60h或者64h)是否空闲，只有当这两个端口空闲的时候我们才可以向其传入数据，等待其空闲的代码为：



在这里testb $0x2, %al是用来检测64h端口是否为空闲，当输入缓存区为空，即我们可以向其传入数据时，64h端口中的状态寄存器的值为0x2，所以我们可以通过这条指令来等待64h端口空闲。在等到64h空闲之后我们会写入0xd1，表明我们要向60h里面写入数据。



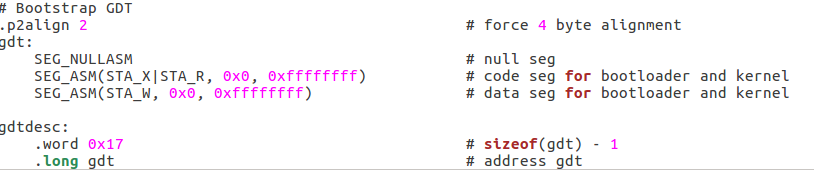
接下来我们需要继续通过64h端口来判断8042芯片是否空闲，在等到空闲之后，我们将0xdf写入60h端口，至此来打开A20开关。



然后我们需要加载GDT全局描述符。在代码里我们看到只用了一句指令便实现了加载GDT的操作：



所以，我们在这里继续跟踪这条指令，来看下这条指令具体是怎么加载GDT全局描述符的。



我们首先追踪gdtdesc可以看到，它里面有两个参数，首先是word 0x17表示的是我们GDT表的大小，其次是long gdt表示的是我们GDT表的入口地址，然后我们可以看到上面便是gdt的定义。根据资料的查询我们可以知道，GDT全局描述符表由三个全局描述符组成，根据规定，第一个均为空描述符，第二个为代码段描述符，第三个为数据段描述符，所以在这里会相应的对应的三个内联汇编用来实现相应的操作，至此，我们的GDT表顺利加载完成。

接下来我们来继续观察对于寄存器CR0的操作：



一共包括了三个操作，首先将cr0寄存器里面的内容取出来，然后进行一个或操作，最后将得到的结果再写入cr0中，由上文我们知道，在这里需要将cr0的最低位设置为1，所以我们的或操作是用来使得cr0的最低位为1的操作，也就是说我们的CR0\_PE\_ON的值必须为1，这样才可以达成目的，然后通过查询CR0\_PE\_ON的定义我们发现的确为1，所以顺利开启PE位。



最后通过一个长跳转指令正式进入保护模式.

