**中山大学计算机学院本科生实验报告**

**（2021学年春季学期）**

课程名称：操作系统  **批改人：**

|  |  |  |  |
| --- | --- | --- | --- |
| 实验 | **Lab1** | 专业（方向） | **计算机科学与技术** |
| 学号 | **18329015** | 姓名 | **郝裕玮** |
| Email | **935087919@qq.com** | 完成日期 | **2021.4.2** |

# 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。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，读者可以了解到：

# 计算机原理

# CPU的编址与寻址: 基于分段机制的内存管理

# CPU的中断机制

# 外设：串口/并口/CGA，时钟，硬盘

# 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

# 外设访问：读硬盘，在CGA上显示字符串

# 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

# 函数调用关系：在汇编级了解函数调用栈的结构和处理过程

# 中断管理：与软件相关的中断处理

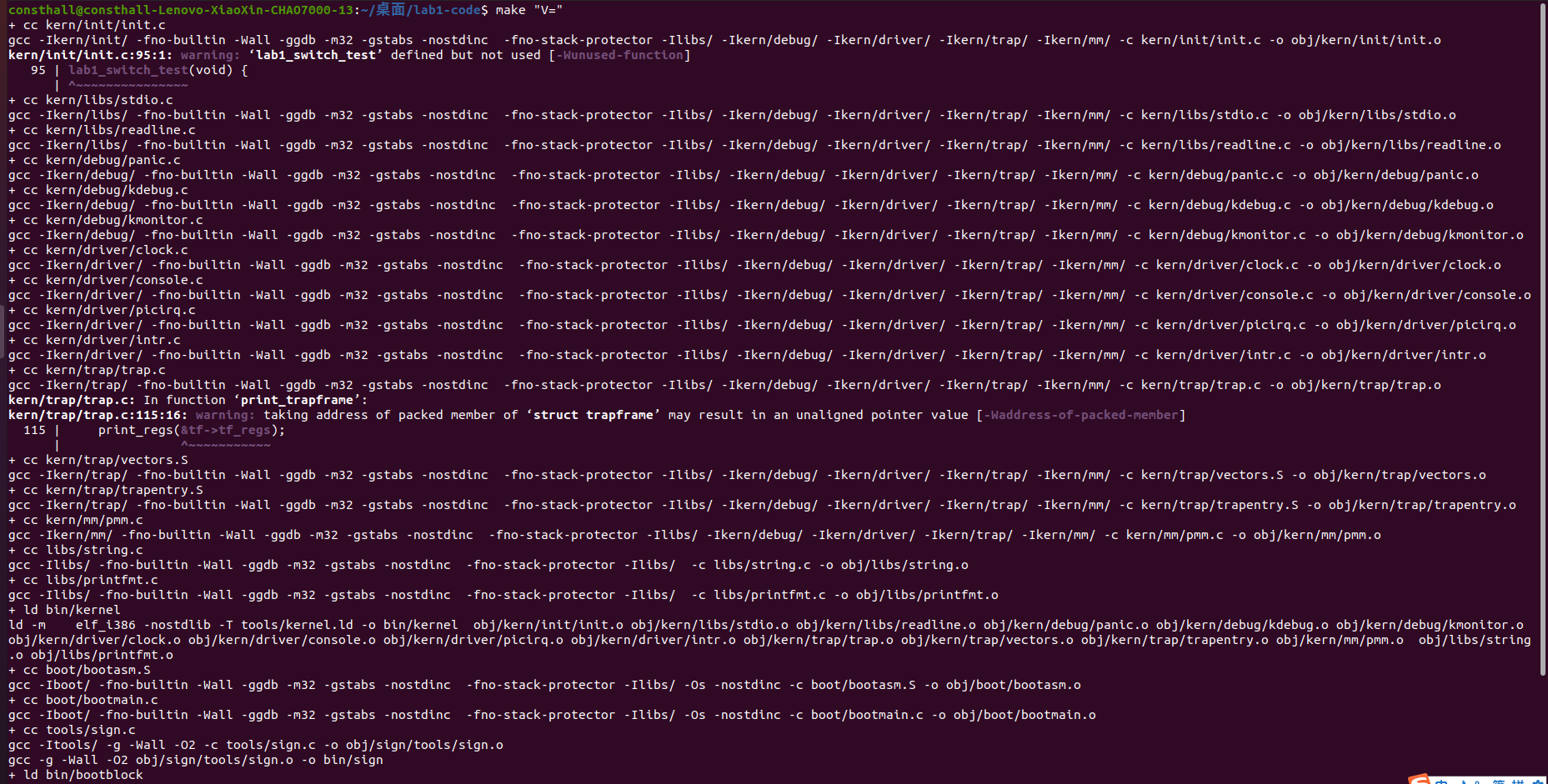
# 外设管理：时钟

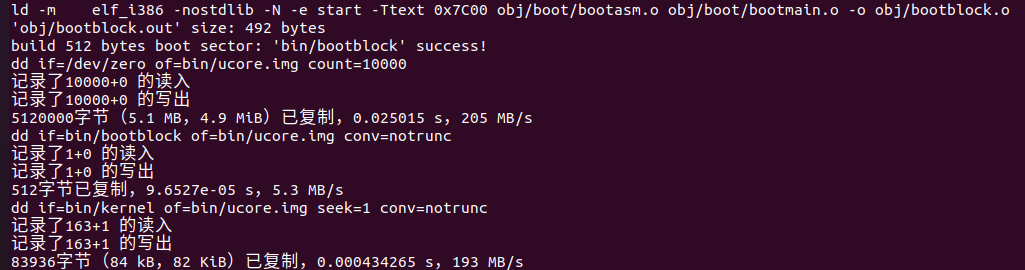
# 实验过程和核心代码及代码解释&实验结果

练习1：理解通过make生成执行文件的过程

1. 操作系统镜像文件ucore.img是如何一步一步生成的？

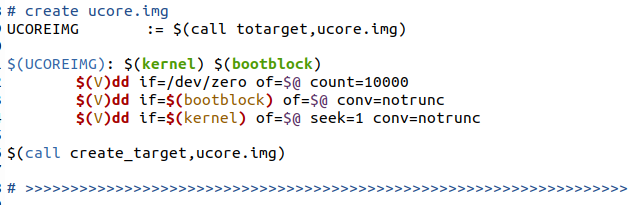
答：首先对Lab1文件执行make”V=”操作，执行结果如下图所示：





之后我们打开Makefile文件进行查看：

生成ucore.img的代码块如下所示：



其中：dd：用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换。

if=文件名：输入文件名，缺省为标准输入。即指定源文件。< if=input file >

of=文件名：输出文件名，缺省为标准输出。即指定目的文件。< of=output file >

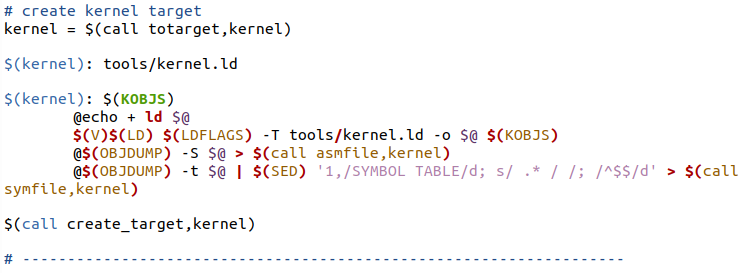
count=blocks：仅拷贝blocks个块，块大小等于ibs指定的字节数。

conv=conversion：用指定的参数转换文件。

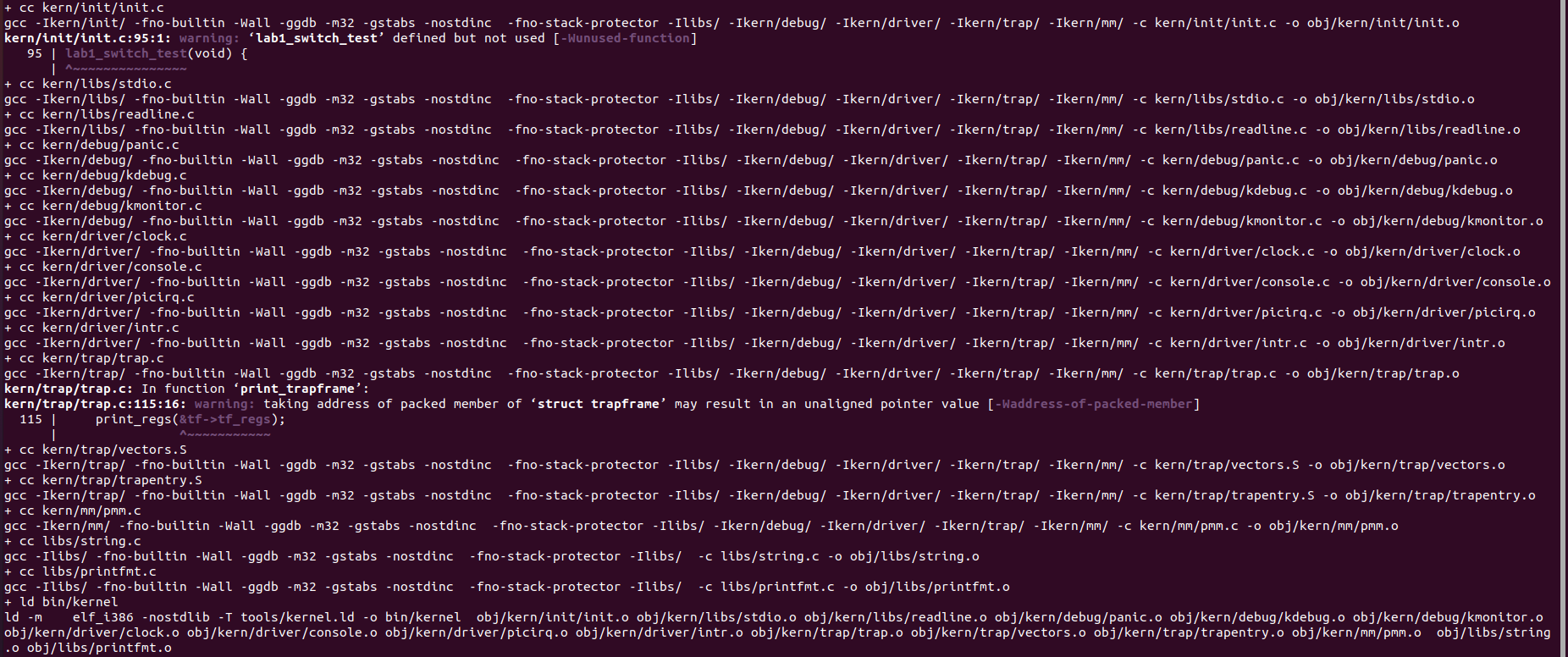
conv=notrunc:不截短输出文件

所以由上述代码易知生成ucore.img需要先生成kernel和bootblock，之后再创建一个10000字节的块，并将bootblock和kernel复制过去，最后即可生成ucore.img。

以下为生成kernel的代码部分：

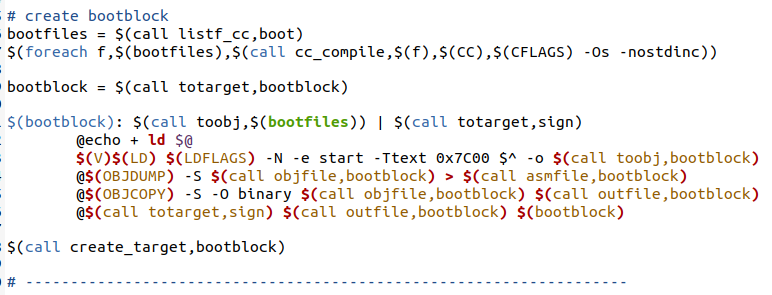


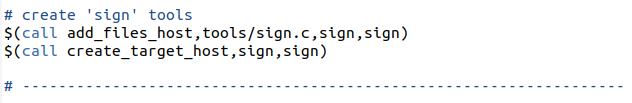
查看Makefile文件，可知需用gcc对kernel目录下的.c文件进行编译：



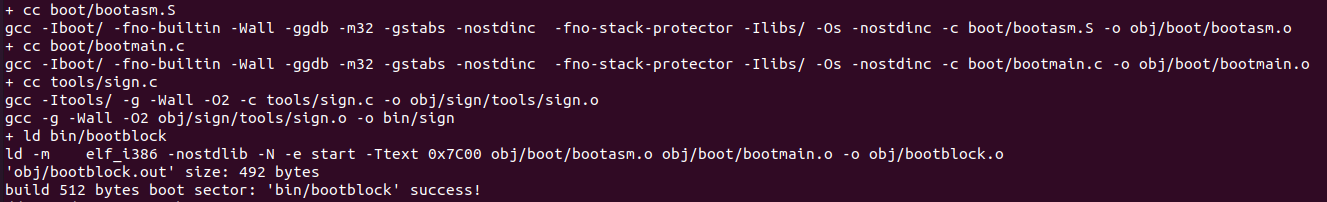
# 上图中的最后一行是用ld命令生成链接程序kernel.ld，并将图中所有生成的.o文件链接成bin/kernel。

以下为生成bootblock和sign的代码部分：





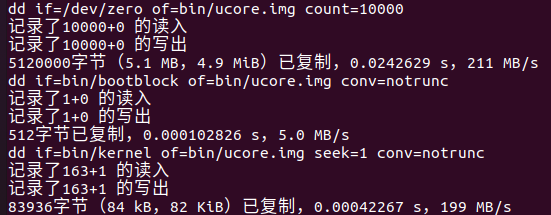
查看Makefile文件，同理可知需用gcc对boot目录下的.c和.s文件以及tools目录下的sign.c文件进行编译：



上图中的第7行是根据sign规范生成bootblock。

上图中的第8行是用ld命令将图中所有生成的.o文件链接成obj/bootblock.o。

以下为最后生成ucore.img的Makefile部分：

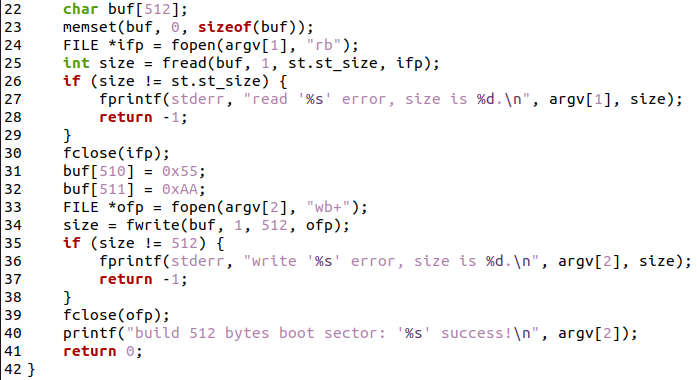


该部分体现的是：创建大小为10000个块的ucore.img，初始化为0，每个块为512字节。

并使用拷贝命令dd将bin/bootblock写入bin/ucore.img的第1个块，之后从bin/ucore.img的第2个块的位置开始将bin/kernel按顺序写入。

2. 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

答：首先我们可先观察tools/sign.c中的部分代码：



（1）由35-37行可知，磁盘主引导扇区大小为512字节；

（2）由31-32行可知，磁盘主引导扇区的最后两个字节的信息是固定的，第510个字节为0x55，第511个字节为0xAA。

练习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。

答：根据附录可知，我们在tools/gdbinit中添加5行代码：

set arch i8086

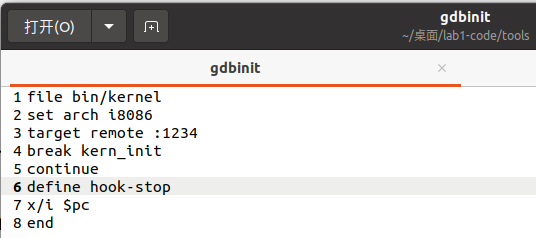
target remote :1234

define hook-stop

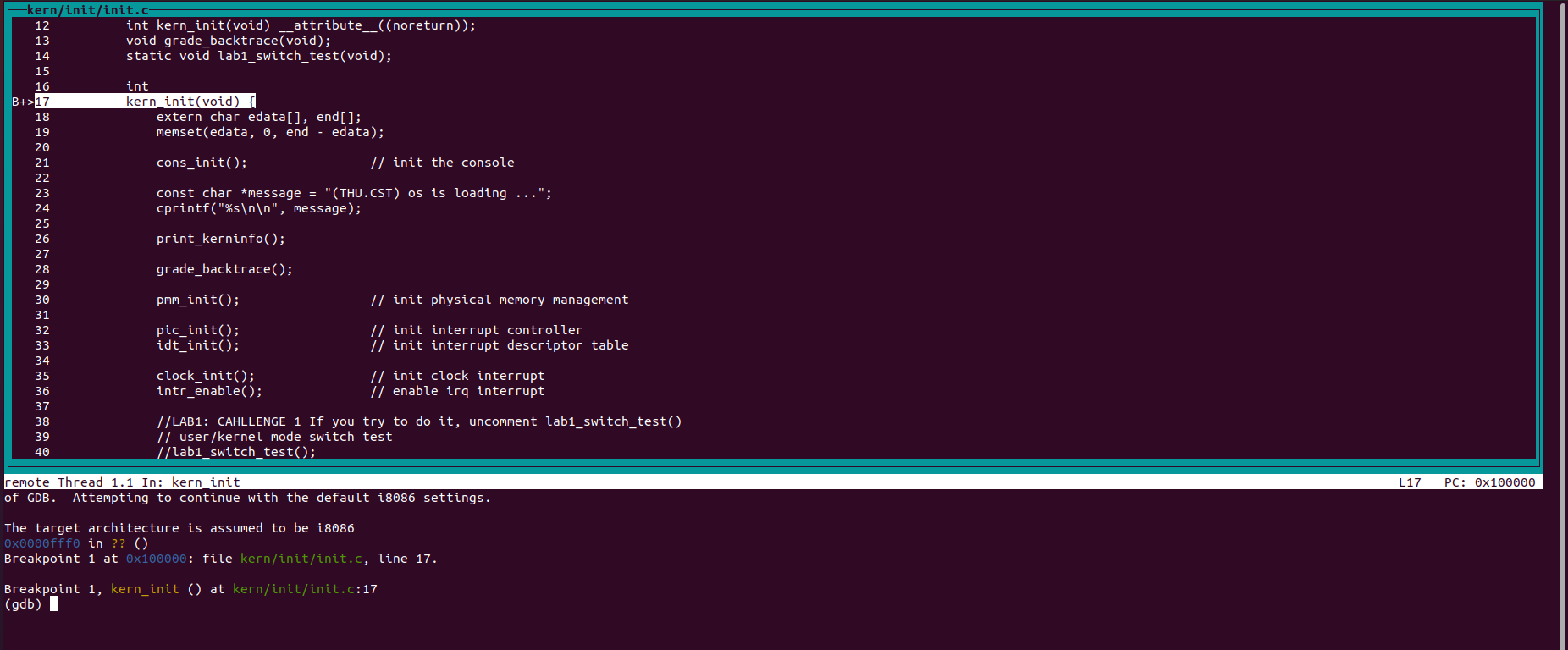
x/i $pc

end

即修改为：



之后在lab1目录下执行make debug：



可见程序断点停在了kern\_init函数入口处，也即CPU加电后执行的第一条指令。

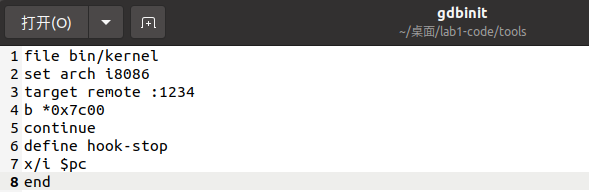
输入si进行单步调试：



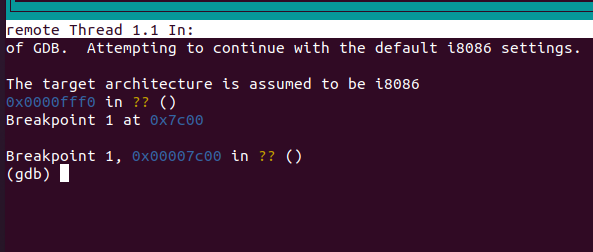
易知单步调试运行正常。

2.在初始化位置0x7c00设置实地址断点,测试断点正常。

答：将tools/gdbinit的内容修改为：



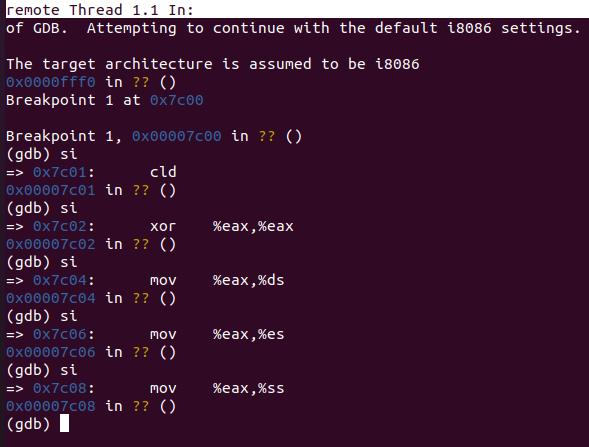
再次在lab1目录下执行make debug：



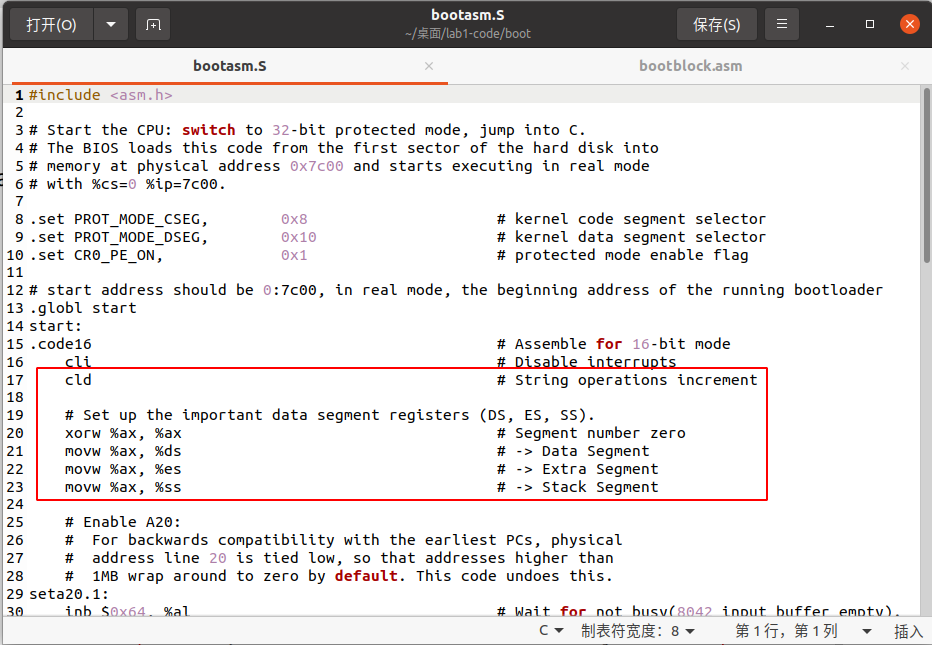
由上图可知，程序停在了0x7c00处，所以实地址断点测试正常。

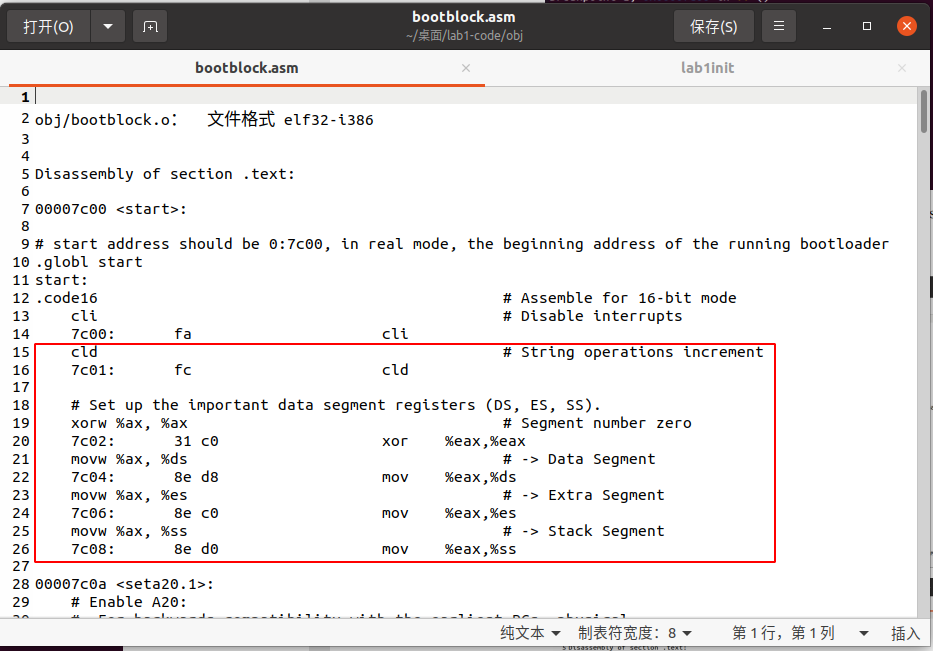
3. 从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和bootblock.asm进行比较。

答：单步追踪断点0x7c00后的五条指令：



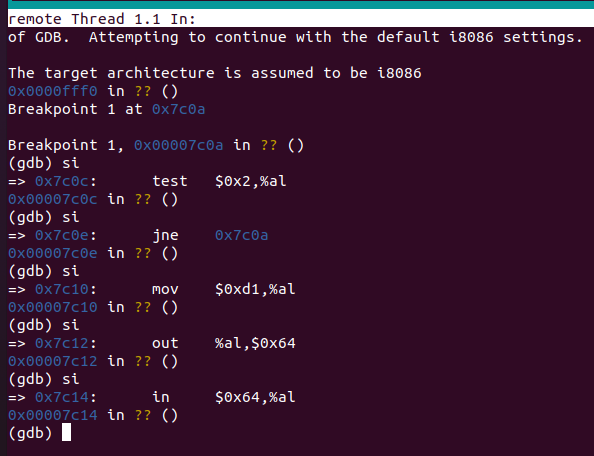
结果如上图所示，发现单步跟踪反汇编得到的代码确实与bootasm.S和bootblock.asm相同。

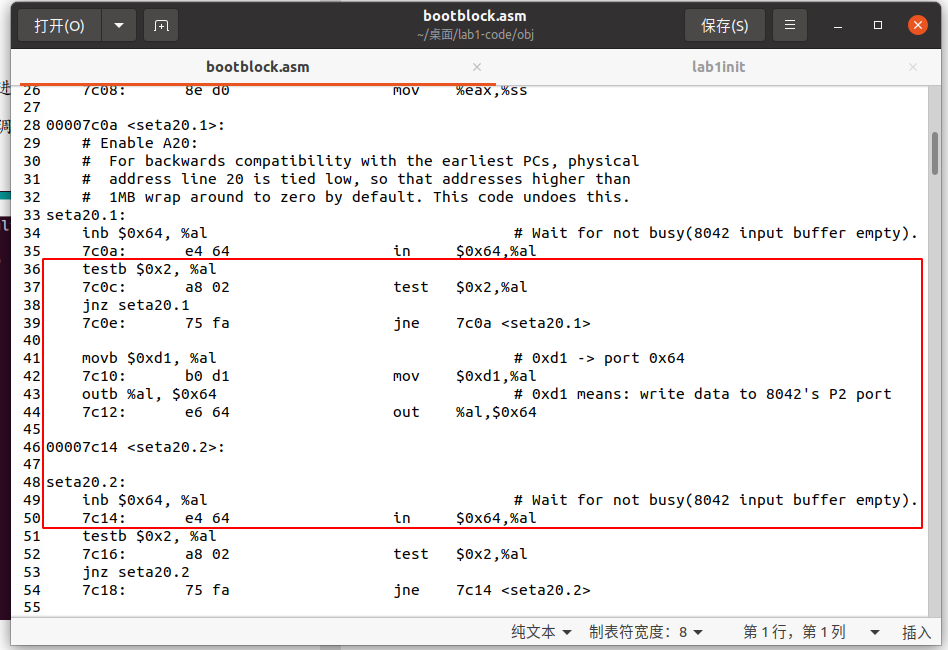


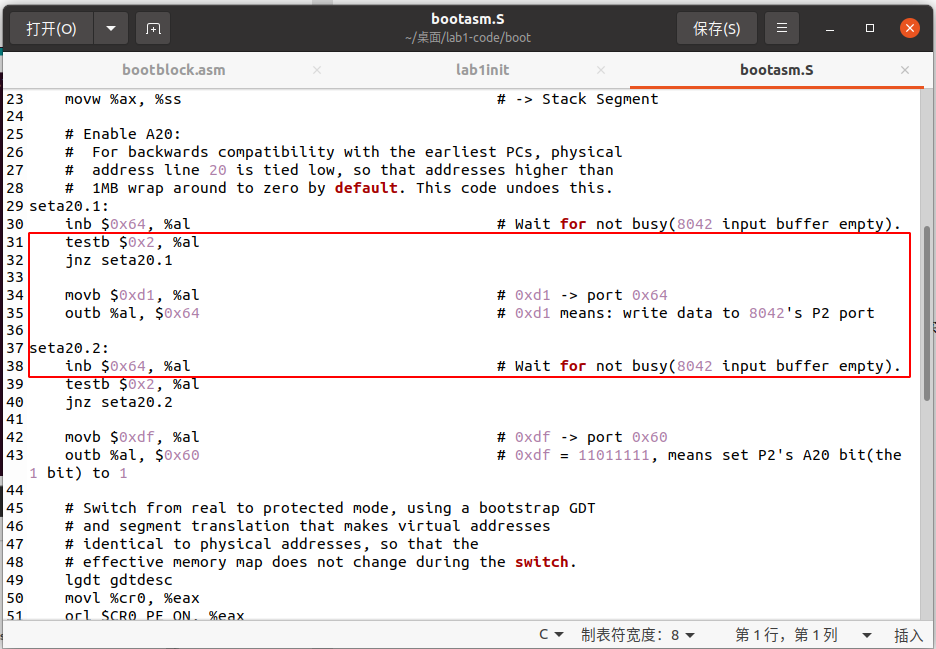


4. 自己找一个bootloader或内核中的代码位置，设置断点并进行测试。

答：设置断点为0x7c0a，并单步追踪接下来5条指令，并将调试结果与bootblock.asm和bootasm.S进行对比：







对比可知该断点测试正常。

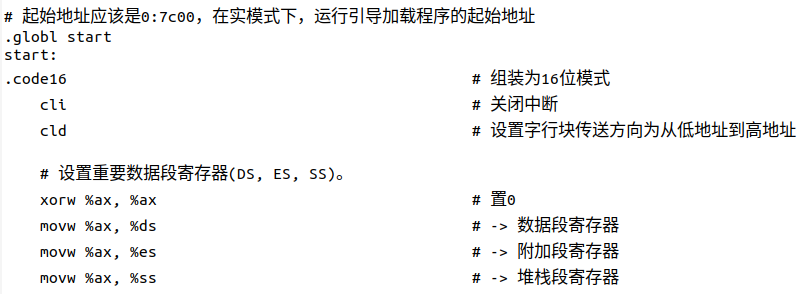
练习3：分析bootloader进入保护模式的过程

答：bootloader进入保护模式的过程可查看bootasm.S的源码（源码中的英文注释已翻译成中文并在某些部分添加了自己对于该部分代码的理解）：

宏定义：

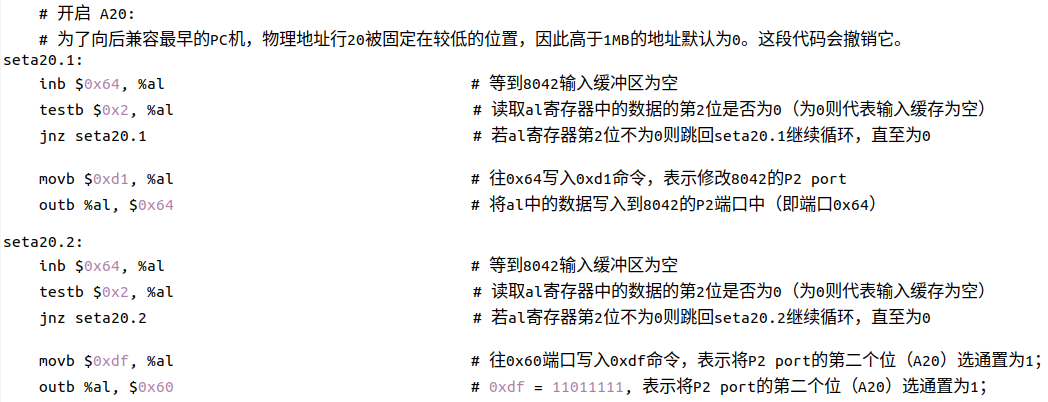


关闭中断，并将各个段寄存器置0（初始化）：



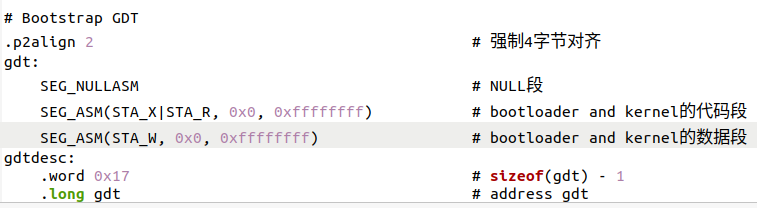
开启A20

开启A20的原因：当 A20 地址线关闭时，所有程序等同于在 8086 中运行。且8086只有20根地址总线，所以在实模式下可访问的物理内存空间不能超过1MB。又因为bootloader需要从实模式切换至保护模式，而在保护模式下需要访问更多，更大的地址空间，所以我们需要将键盘控制器上的A20线置于高电位，使得全部的32条地址线可用，这样就使得内存空间从之前的1M变成了4G。



初始化GDT表：

查看bootasm.S可知代码中已经静态的描述了一个简单的GDT，如下图所示：

  
 所以开启A20之后，我们调用命令lgdt gdtdesc即可载入全局描述符表。

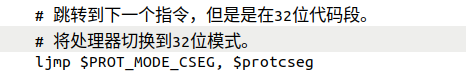


进入保护模式

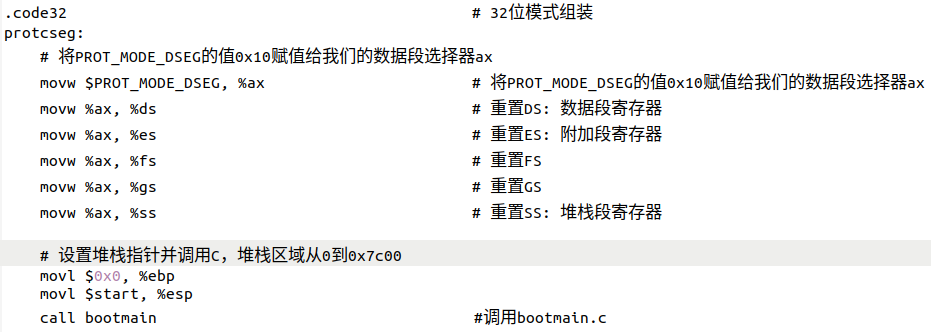
（1）将cr0置1开启保护模式（CR0\_PE\_ON在宏定义中已被预赋值为1）



（2）使用长跳转指令ljmp，将cs修改为32位段寄存器，并跳转到protcseg这一32位代码入口处，此时CPU进入32位模式：



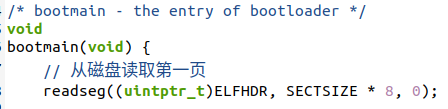
（3）重置保护模式下的段寄存器并建立堆栈，最后调用bootmain.c，进行操作系统内核的加载。至此，bootloader已经完成了从实模式进入到保护模式的任务。



练习4：分析bootloader加载ELF格式的OS的过程

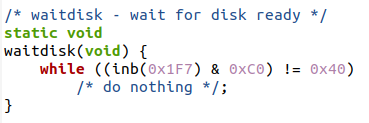
（1）bootloader如何读取硬盘扇区的？

答：bootmain函数中调用readseg函数读取硬盘扇区



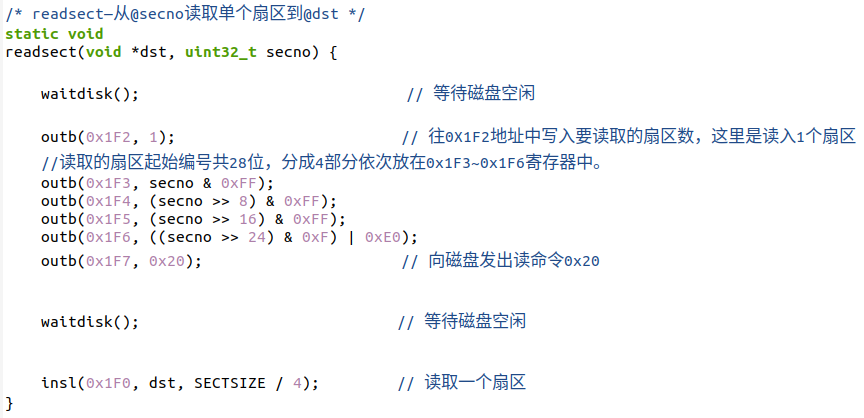
因为readseg函数中调用了readsect函数，而readsect函数又调用了waitdisk函数。所以我们先按照waitdisk，readsect，readseg的顺序对这三个函数进行解释：

对于waitdisk函数：



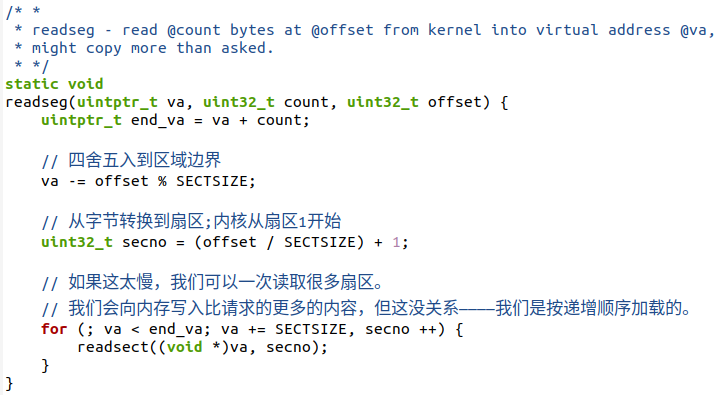
该函数功能为：不断查询读0x1F7寄存器的最高两位，直到最高位为0、次高位为1（即磁盘空闲）才返回。

对于readsect函数：



该函数功能为：读取一个磁盘扇区。

对于readseg函数：



该函数功能为：从扇区1起的第offset个位置处，读取count个字节到指定内存中。

此外，va -= offset % SECTSIZE表示：就算读取了整个扇区，这样操作就可以使得读到的数据在内存中的起始位置仍然是参数va。

（2）bootloader是如何加载ELF格式的OS？

答：ELF简介：ELF(Executable and linking format)文件格式是Linux系统下的一种常用目标文件(object file)格式，有三种主要类型:

用于执行的可执行文件(executable file)，用于提供程序的进程映像，加载的内存执行。 这也是本实验的OS文件类型。

用于连接的可重定位文件(relocatable file)，可与其它目标文件一起创建可执行文件和共享目标文件。

共享目标文件(shared object file),连接器可将它与其它可重定位文件和共享目标文件连接成其它的目标文件，动态连接器又可将它与可执行文件和其它共享目标文件结合起来创建一个进程映像。

ELF结构定义如下代码所示：

struct elfhdr {

uint32\_t e\_magic; // 判断读出来的ELF格式的文件是否为正确的格式

uint8\_t e\_elf[12];

uint16\_t e\_type; // 1=可重定位，2=可执行，3=共享对象，4=核心映像

uint16\_t e\_machine; // 3=x86，4=68K等.

uint32\_t e\_version; // 文件版本，总是1

uint32\_t e\_entry; // 程序入口所对应的虚拟地址。

uint32\_t e\_phoff; // 程序头表的位置偏移

uint32\_t e\_shoff; // 区段标题或0的文件位置

uint32\_t e\_flags; // 特定于体系结构的标志，通常为0

uint16\_t e\_ehsize; // 这个elf头的大小

uint16\_t e\_phentsize; // 程序头中条目的大小

uint16\_t e\_phnum; // 程序头表中的入口数目

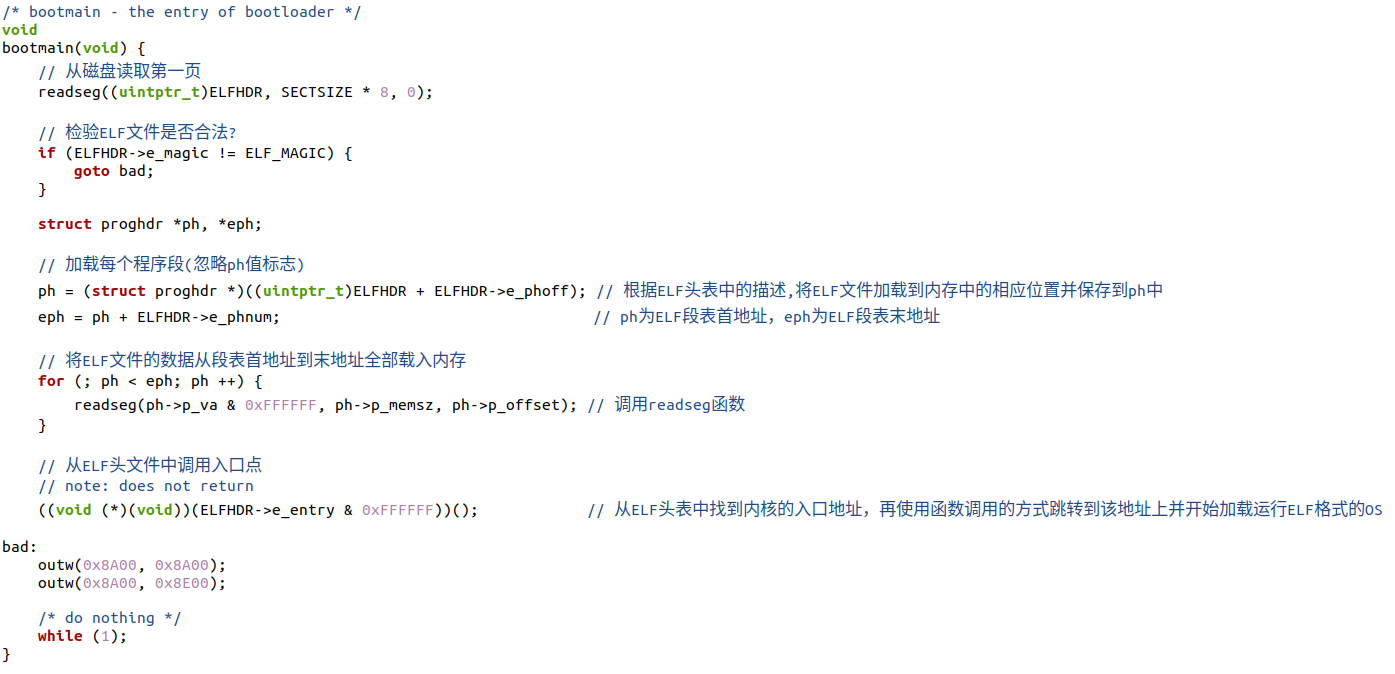
uint16\_t e\_shentsize; // 节标题中条目的大小

uint16\_t e\_shnum; // 节标题中的条目数或0

uint16\_t e\_shstrndx; // 包含节名称字符串的节号。

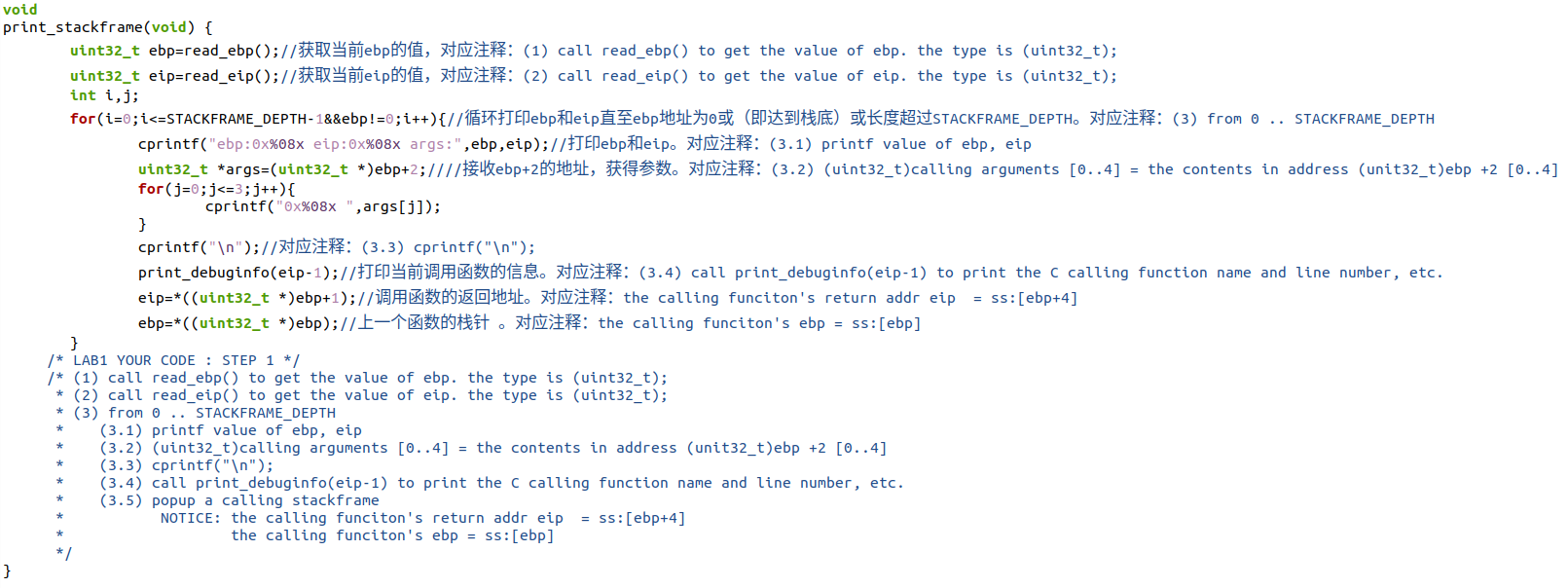
};

具体加载过程详见如下bootmain函数代码注释：

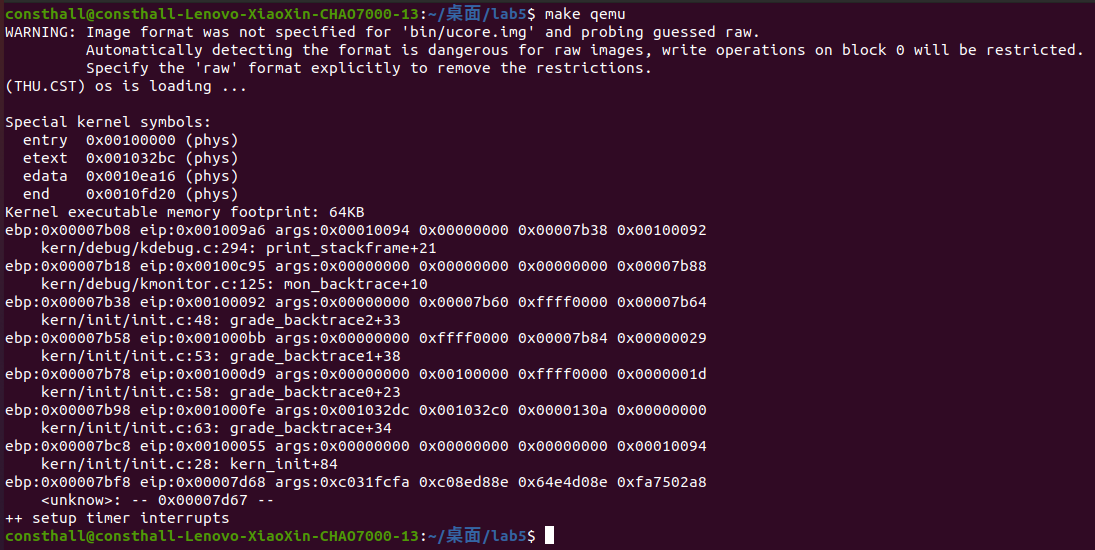


练习5：实现函数调用堆栈跟踪函数

答：根据kdebug.c中print\_stackframe函数部分的注释即可写出该部分代码（对于补充代码的解释与分析已放在每一行的注释中）：



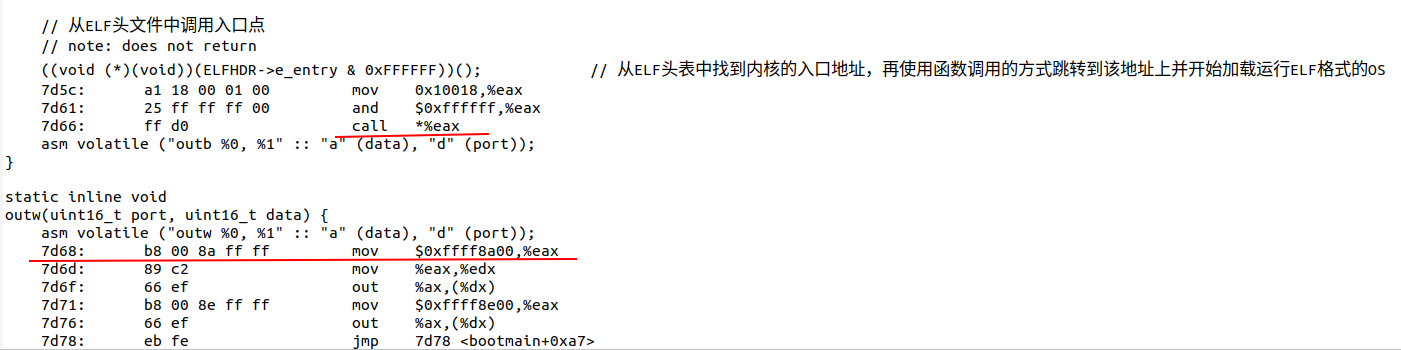
运行结果如下所示：



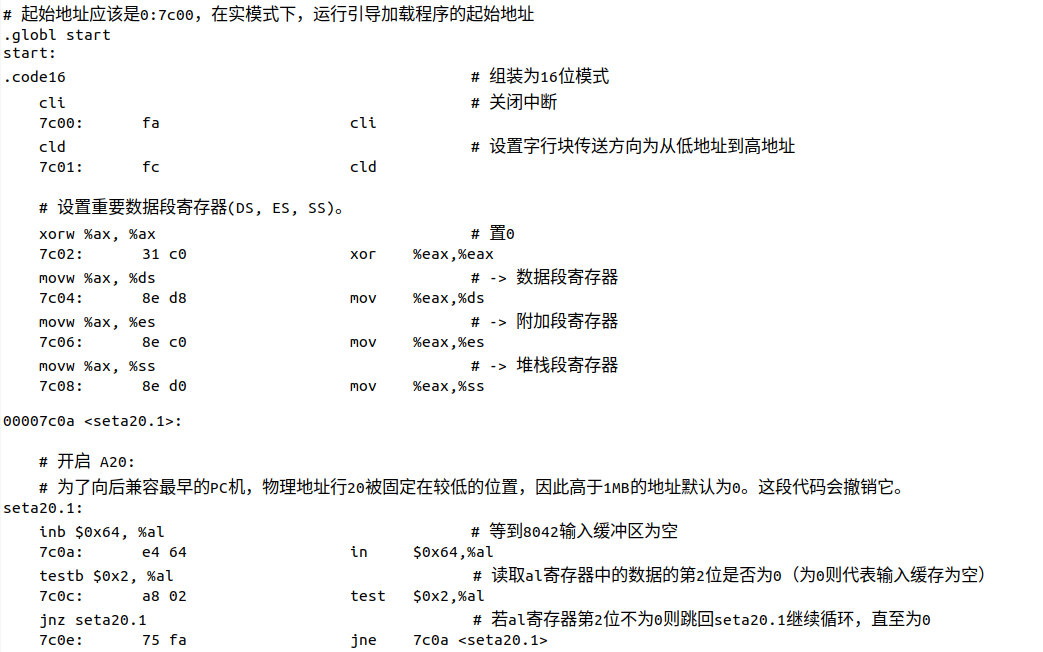
最后一行各个数值的含义：

ebp:0x00007bf8 ：最后一行的ebp的值是kern\_init函数的栈顶地址；

eip:0x00007d68：最后一行的eip的值是kern\_init函数的返回地址，即bootmain函数调用kern\_init对应指令的下一条指令的地址（详见下图）。



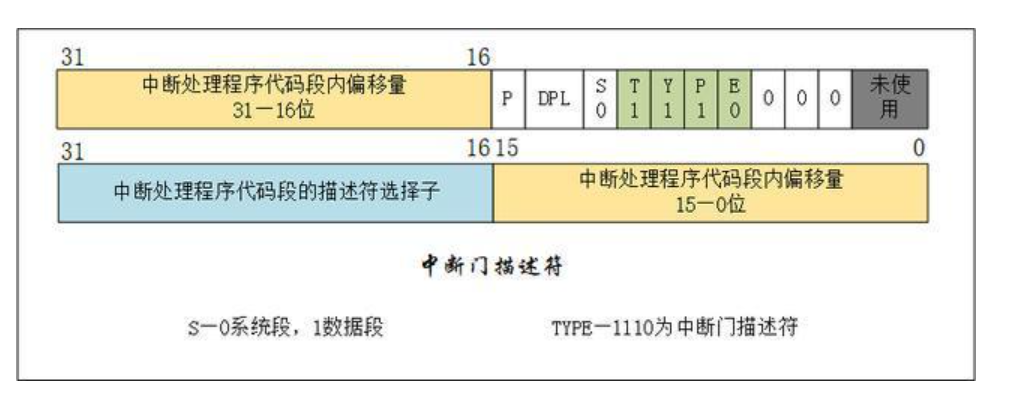
args：通常情况下，args表示被调用函数的参数。但是这里不同（因为bootmain函数不需要参数），在经过与bootblock.asm的对比之后可发现args的四个值对应着bootloader指令的前16个字节（详见下图对比）：



<unknow> - - 0x00007d67 - -：bootmain函数内调用OS kernel入口函数的该指令的地址

练习6：完善中断初始化和处理 （需要编程）

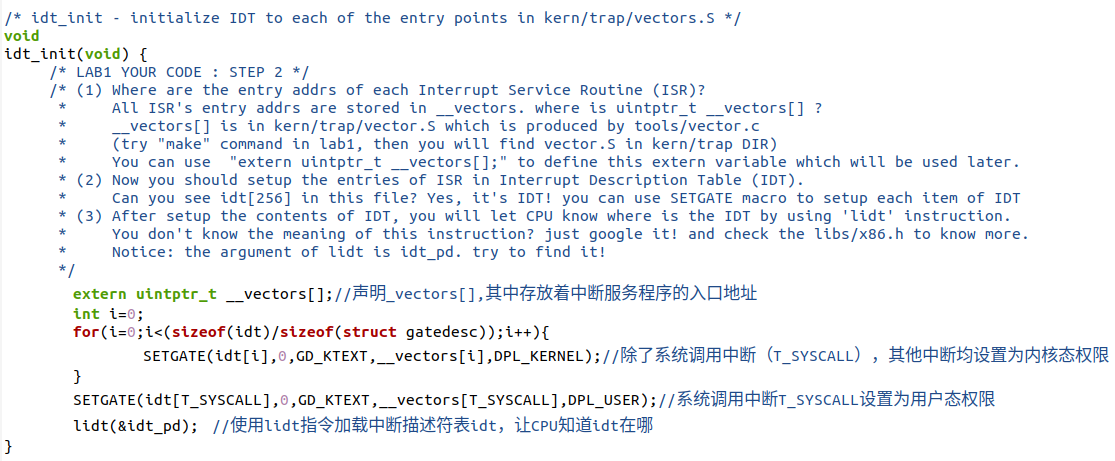
1. 中断描述符表（也可简称为保护模式下的中断向量表）中一个表项占多少字节？其中哪几位代表中断处理代码的入口？



答：IDT中一个表项占8个字节。其中0-15位是偏移量offset的低16位，16-31位是处理代码入口地址的段选择子，48-63位是偏移量offset的高16位。0-15位和48-63位结合即可得到段内偏移量，再由段选择子得到段基址，最后将段基址和段内偏移量相加即可得中断处理代码的入口。

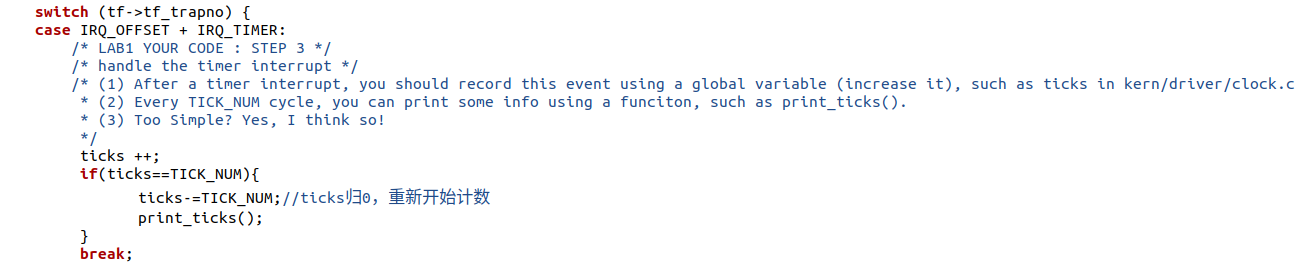
1. 请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt\_init。在idt\_init函数中，依次对所有中断入口进行初始化。使用mmu.h中的SETGATE宏，填充idt数组内容。每个中断的入口由tools/vectors.c生成，使用trap.c中声明的vectors数组即可。

答：根据idt\_init中的注释即可补全代码，代码及其注释详见下图：

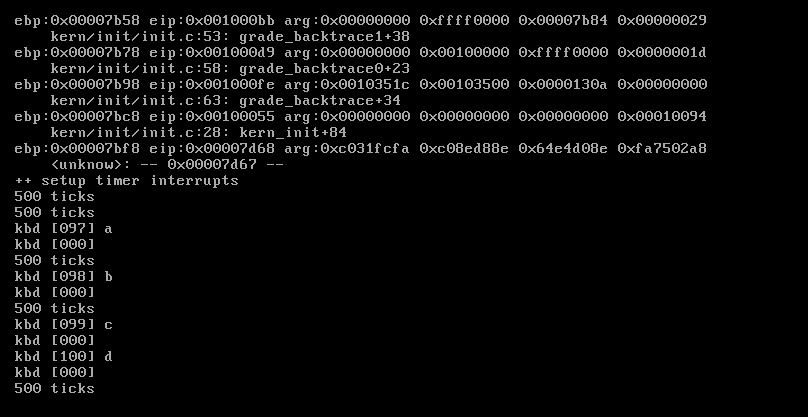


（3）请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写trap函数中处理时钟中断的部分，使操作系统每遇到500次时钟中断后，调用print\_ticks子程序，向屏幕上打印一行文字”500 ticks”。

答：根据trap\_dispatch函数中的case IRQ\_OFFSET + IRQ\_TIMER中的注释即可补全代码，代码及其注释详见下图（其中TICK\_NUM已在宏定义中声明为500）：



运行结果见下图：



如结果所示每5秒出现一次500 ticks，且在运行过程中按下A、B、C、D键也会显示在屏幕上。

# 实验感想

本次实验是操作系统的第一次实验，一开始我还是很不适应的，因为感觉到实验中的内容相较于理论课更加深入，感觉到无从下手。

我在参考了网上的几篇相关实验报告之后，才大致有了思路和实验方向。虽然有一些部分参考了他人的成果，但我都是在学习过后自己独立写出实验报告和相关代码，并且在代码当中补充上了很多自己的注释，以表明自己实实在在的弄懂了这些问题。

本次实验主要有以下一些问题：

对于makefile的原理和qemu的调试不熟，经过对老师所发放的参考资料的学习和针对自己遇上的问题在网上进行查阅之后均得到解决。

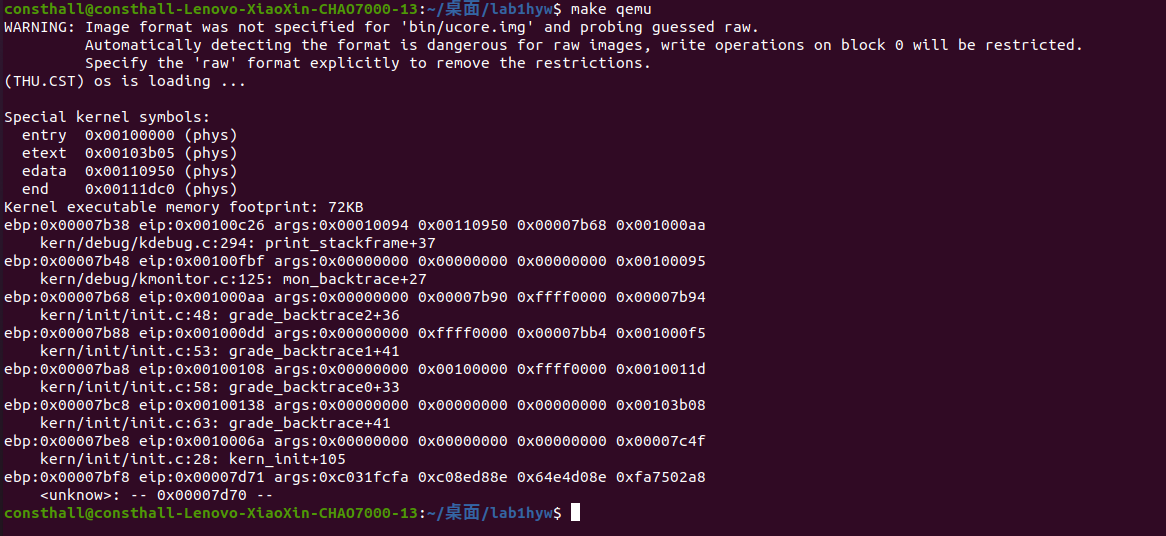
对于练习3，4这一类原理解释性题目束手无措，所幸bootasm.S和bootmain.c的书写语言是汇编和C，文件中的大部分代码还是以解释性语言为主，所以结合注释也能大致看懂代码的作用。在仔细阅读源码和对比参考资料之后这两个练习圆满完成。

该问题在实验完成后仍未解决，网上查阅相关资料也未找到答案，最后在咨询过黄聃老师后得到解决：

由于之前我提前安装了双系统和最新版的qemu，所以此次实验我并没有安装虚拟机，直接从超算习堂上下载lab1的相关文件后就开始进行实验。

在前4个练习中并没有出现异样，但是从第5个练习开始，我的运行结果和使用虚拟机的同学出现了偏差。

对于练习5，我在补充完kdebug.c代码部分之后在终端运行make qemu,发现最后一行没有出现++ setup timer interrupts（即下图结果）（实验报告图中的结果有，结果截图的来源会在后面补充），但是当时我并没有意识到这一问题，继续进行练习6的实验。



对于练习6，我在补充完trap.c代码部分之后在终端运行make qemu，发现结果和练习5保持一致，没有任何变化，在多次修改代码并且和网上代码一一对比之后，结果仍没有改变。此时我同学把他虚拟机上已经make过的整个lab1文件夹发给了我，我运行了一下之后发现结果正确，于是我以为可能是老师在超算习堂上上传的文件相对于虚拟机里的文件有差别，就将文件一一替换之后重新make，发现结果又回到了和练习5一样。

在排查之后我发现不是文件问题，即如果我直接运行我同学发来的文件，不make clean且重新make，那么结果就是正确的，反之结果就错误。为了验证我的想法，我把我的lab1文件发给了我同学，让他在虚拟机上make clean再 make，make qemu之后发现我的文件在虚拟机上结果运行正确。

我在网上查询了双系统和虚拟机在make上的区别，并没有查到有用的回答。所以对于练习5，6，我只能将我的文件在我同学的虚拟机上make运行之后再重新发回到双系统上直接make qemu查看结果。

在咨询过黄聃老师后，黄聃老师给出的回答是：应该是我的ubuntu和gcc版本和虚拟机不一致，因为ucore这个代码是2015年以前的代码，如果想要ubuntu的编译运行结果和虚拟机保持一致，则需要保证操作系统，kernel版本，gcc版本，和虚拟机中保持一致。

所以为了方便起见，我决定从下次实验开始用虚拟机进行实验，从而避免不必要的麻烦。