**Ucore实验lab1报告**

1. **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：

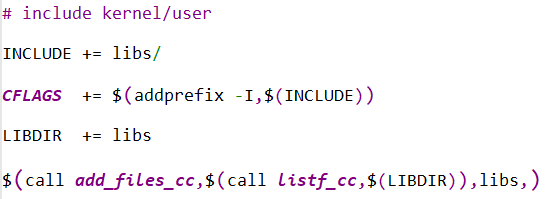
* 计算机原理
  + CPU的编址与寻址: 基于分段机制的内存管理
  + CPU的中断机制
  + 外设：串口/并口/CGA，时钟，硬盘
* Bootloader软件
  + 编译运行bootloader的过程
  + 调试bootloader的方法
  + PC启动bootloader的过程
  + ELF执行文件的格式和加载
  + 外设访问：读硬盘，在CGA上显示字符串
* ucore OS软件
  + 编译运行ucore OS的过程
  + ucore OS的启动过程
  + 调试ucore OS的方法
  + 函数调用关系：在汇编级了解函数调用栈的结构和处理过程
  + 中断管理：与软件相关的中断处理
  + 外设管理：时钟

1. **实验内容**

#### 练习一：理解通过make生成执行文件的过程

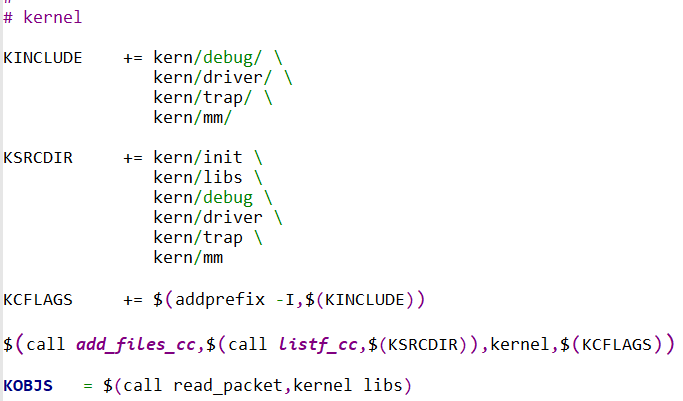
1.操作系统镜像文件ucore.img是如何一步一步生成的？(需要比较详细地解释Makefile中每一条相关命令和命令参数的含义，以及说明命令导致的结果)

1. 编译libs和kern目录下所有的.c和.S文件，生成.o文件，并链接得到bin/kernel文件

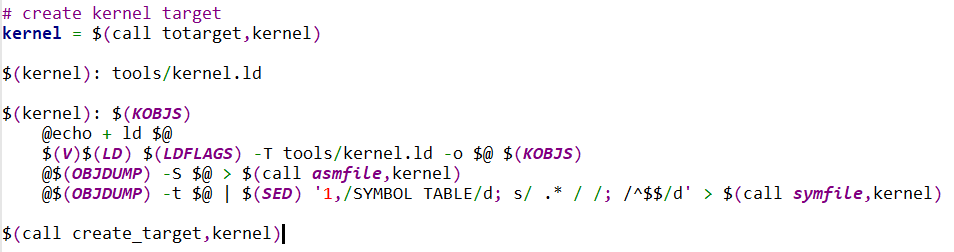


listf\_cc = $(call listf,$(1),$(CTYPE)) 中调用了listf函数，调用时传入的第1个参数为$(1) = $(LIBDIR) = libs，第2个参数为$(CTYPE) = c S。

listf函数的定义为listf = $(filter $(if $(2),$(addprefix %.,$(2)),%), $(wildcard $(addsuffix $(SLASH)\*,$(1))))，调用listf的返回结果为libs目录下的所有.c和.S文件。由于lab1的libs目录下只有.h和.c文件，因此最终返回.c文件。



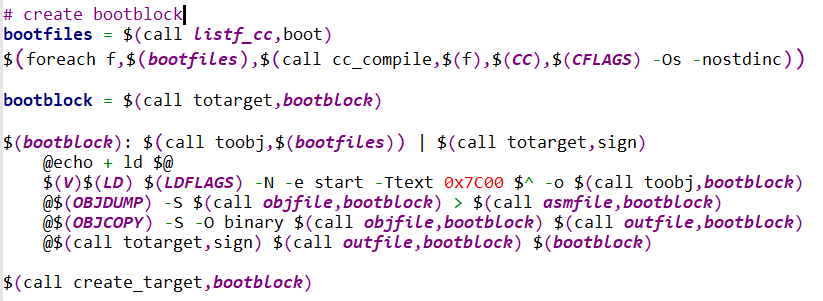
生成kern目录下的obj文件名，过程和上面类似。



kernel = $(call totarget,kernel) 设置生成的kernel目标名为bin/kernel

$(kernel): tools/kernel.ld 指出kernel目标文件需要依赖tools/kernel.ld文件，而kernel.ld文件是一个链接脚本，其中设置了输出的目标文件的入口地址及各个段的一些属性，包括各个段是由输入文件的哪些段组成、各个段的起始地址等。

1. 编译boot目录下所有的.c和.S文件，生成.o文件，并链接得到bin/bootblock.out文件



$(foreach f,$(bootfiles),$(call cc\_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc)) 编译bootfiles生成.o文件

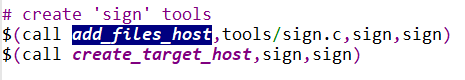
$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign) 声明bin/bootblock依赖于obj/boot/\*.o 和bin/sign文件。toobj函数的作用是给输入参数增加前缀obj/，并将文件后缀名改为.o

$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock) 链接所有.o文件以生成obj/bootblock.o。链接选项中的-e start -Ttext 0x7C00，大致意思是设置bootblock的入口地址为start标签，并且 start标签的地址为0x7C00.

之后通过一系列转换得到obj/bootblock.asm、obj/bootblock.out、bin/bootblock。

1. 编译tools/sign.c文件，得到bin/sign文件

$(call totarget,sign)



add\_files\_host函数调用了add\_files函数，add\_files又调用了do\_add\_files\_to\_packet

do\_add\_files\_to\_packet的作用是生成obj文件，因此这里调用add\_files的作用是设置objs\sign = obj/sign/tools/sign.o

create\_target\_host = $(call create\_target,$(1),$(2),$(3),$(HOSTCC),$(HOSTCFLAGS))

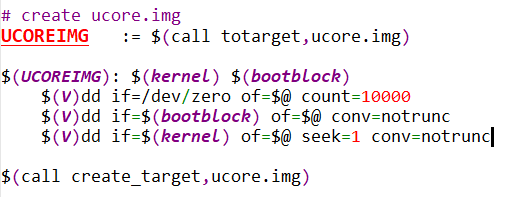
create\_target = $(eval $(call do\_create\_target,$(1),$(2),$(3),$(4),$(5)))

do\_create\_target的作用是生成目标文件，也就是生成obj/sign/tools/sign.o

1. 利用bin/sign工具将bin/bootblock.out文件转化为512字节的bin/bootblock文件，并将bin/bootblock的最后两个字节设置为0x55AA

Sign工具的作用

1. 为bin/ucore.img分配5000MB的内存空间，并将bin/bootblock复制到bin/ucore.img的第一个block，紧接着将bin/kernel复制到bin/ucore.img第二个block开始的位置



$(V)dd if=/dev/zero of=$@ count=10000。这里为bin/ucore.img分配10000个block的内存空间，并全部初始化为0。由于没指定block的大小，因此为默认值512字节，则总大小为5000M。

$(V)dd if=$(bootblock) of=$@ conv=notrunc。这里将bin/bootblock复制到bin/ucore.img

$(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc 继续将bin/kernel复制到bin/ucore.img。选项seek=1的意思是复制时跳过bin/ucore.img的第一个block，从第2个block也就是第512个字节后面开始拷贝bin/kernel的内容。

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？

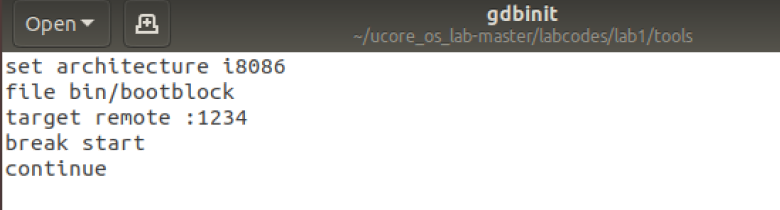
（1）大小为512字节

（2）最后两个字节为0x55AA

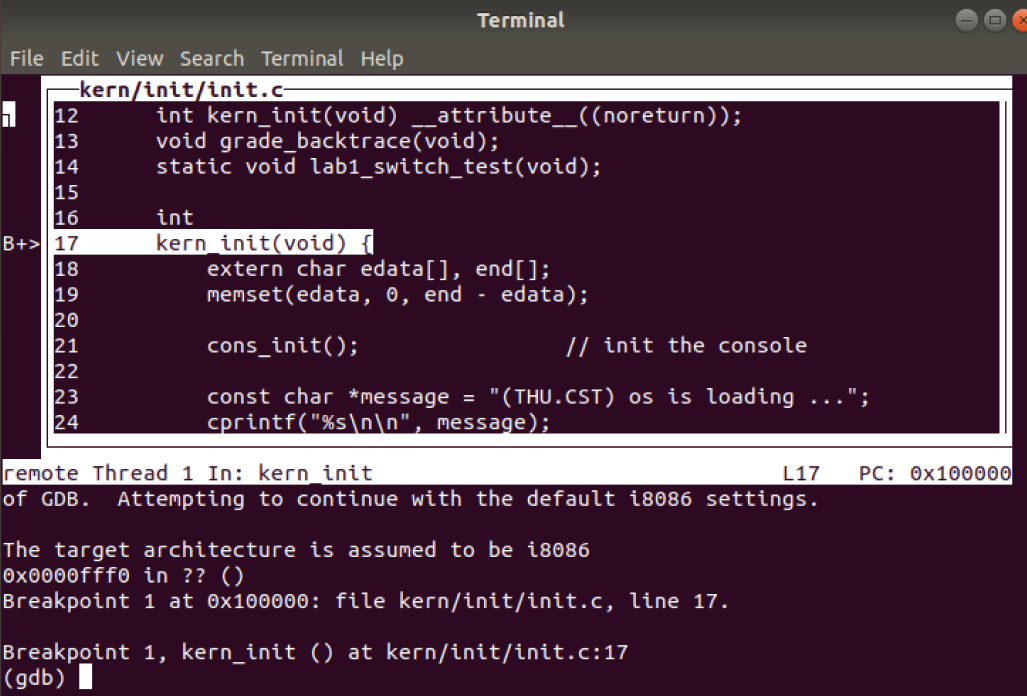
#### 练习二：使用qemu执行并调试lab1中的软件

1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

为了从CPU加电后执行的第一条指令开始，需要修改一下tools/gdbinit



执行make debug



1. 在初始化位置0x7c00设置实地址断点,测试断点正常。

由于boot loader的入口为start，其地址为0x7c00,所以设置为break start

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。

反汇编的代码中的指令不带指示长度的后缀，而bootasm.S的指令则有。

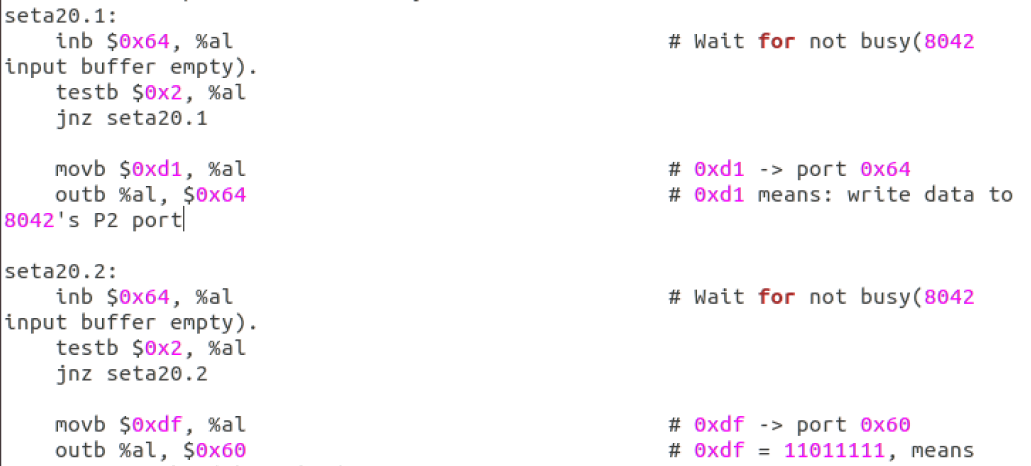
反汇编的代码中的通用寄存器是32位（带有e前缀），而bootasm.S的代码中的通用寄存器是16位（不带e前缀）。

#### 练习三：分析bootloader进入保护模式的过程

1. 为何开启A20，以及如何开启A20

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，

可以访问4G的内存空间。



seta20.1: # 等待8042键盘控制器不忙

movb $0xd1, %al # 发送写8042输出端口的指令

seta20.1: # 等待8042键盘控制器不忙

movb $0xdf, %al # 打开A20

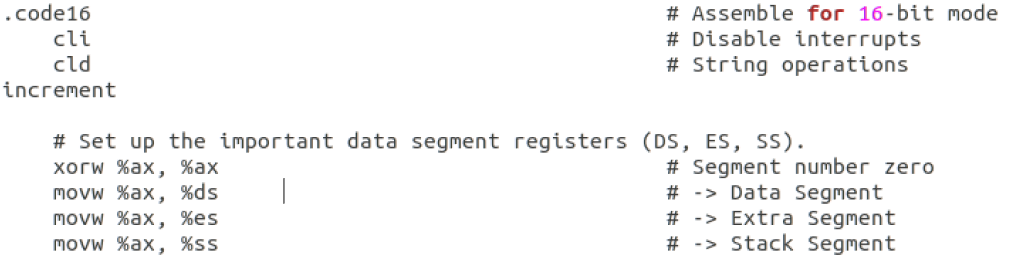
1. 如何初始化GDT表

初始化GDT表：一个简单的GDT表和其描述符已经静态储存在引导区中，载入即可

lgdt gdtdesc

3.如何使能和进入保护模式

分析bootloader 进入保护模式的过程。

从`%cs=0 $pc=0x7c00`，进入后

首先清理环境：包括将flag置0和将段寄存器置0

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

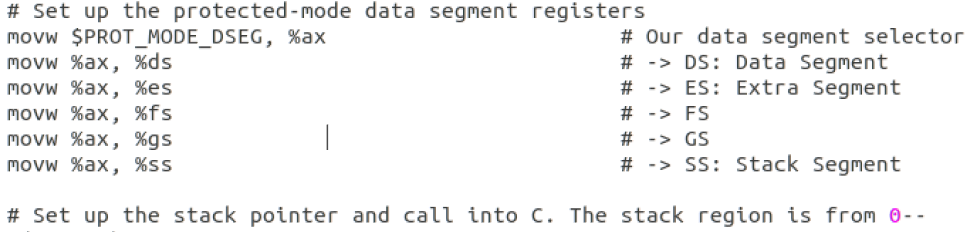
movl %eax, %cr0

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

ljmp $PROT\_MODE\_CSEG, $protcseg

.code32

通过长跳转更新cs的基地址



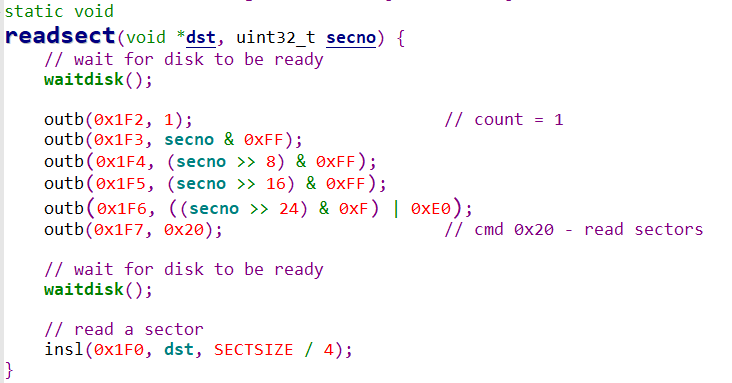
设置段寄存器，并建立堆栈

call bootmain

转到保护模式完成，进入boot主方法

**练习四：分析bootloader加载ELF格式的OS的过程。**

* 1. bootloader如何读取硬盘扇区的？

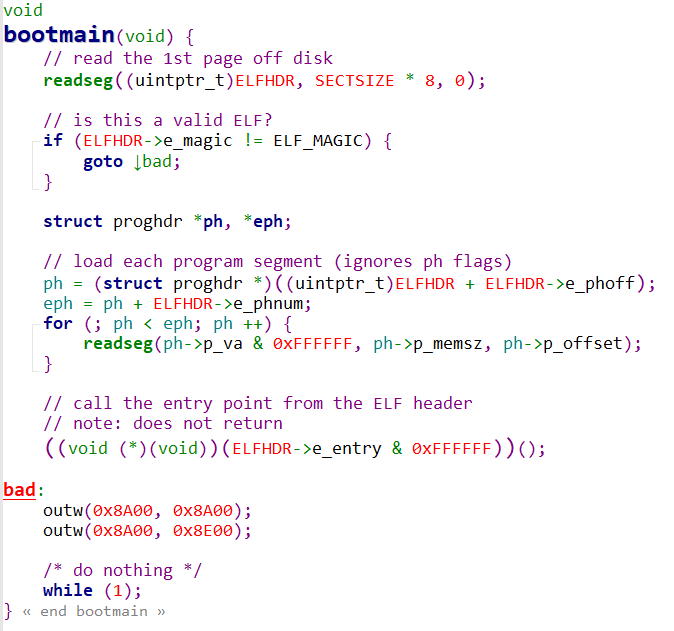


实现bootloader读取扇区的功能

读一个扇区的大致流程：

1. 等待磁盘准备好
2. 发出读取扇区的命令
3. 等待磁盘准备好
4. 把磁盘扇区数据读到指定内存

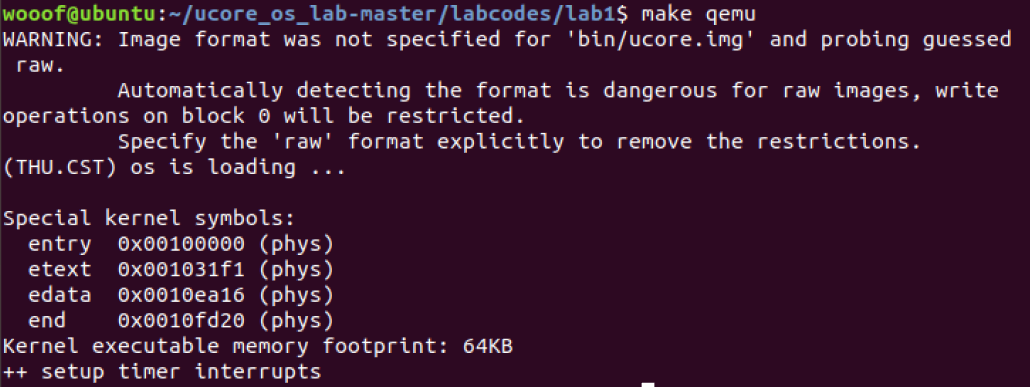
* 2. bootloader是如何加载ELF格式的OS？



1. 首先从硬盘中将bin/kernel文件的第一页内容加载到内存地址为0x10000的位置，目的是读取kernel文件的ELF Header信息。
2. 校验ELF Header的e\_magic字段，以确保这是一个ELF文件
3. 读取ELF Header的e\_phoff字段，得到Program Header表的起始地址；读取ELF Header的e\_phnum字段，得到Program Header表的元素数目。
4. 遍历Program Header表中的每个元素，得到每个Segment在文件中的偏移、要加载到内存中的位置（虚拟地址）及Segment的长度等信息，并通过磁盘I/O进行加载
5. 加载完毕，通过ELF Header的e\_entry得到内核的入口地址，并跳转到该地址开始执行内核代码

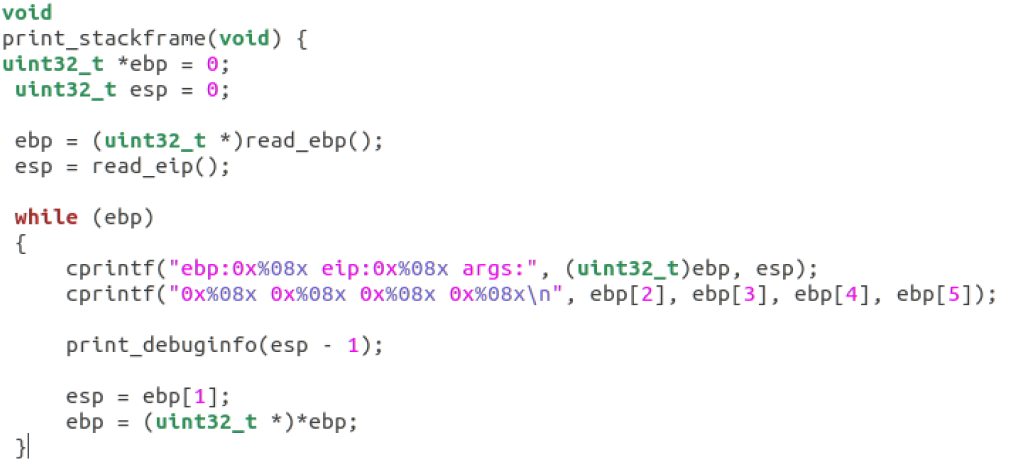
#### 练习五： 实现函数调用堆栈跟踪函数

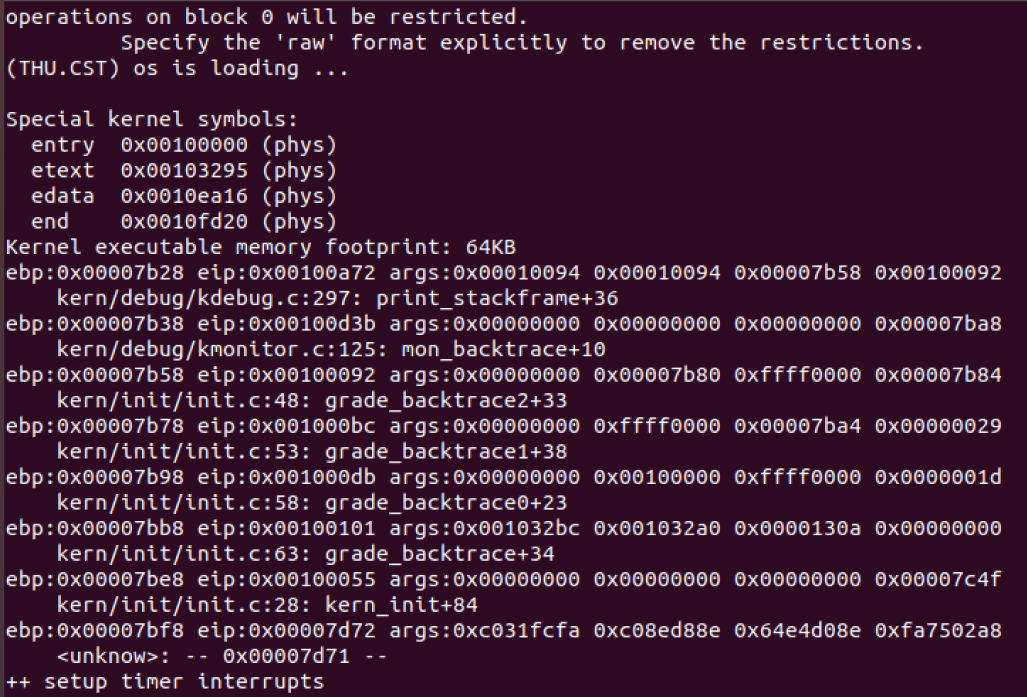
#### 在修改kern/debug/kdebug.c::print\_stackframe函数前进行make qemu

****

对kern/debug/kdebug.c::print\_stackframe函数进行修改

* 首先定义两个局部变量ebp、esp分别存放ebp、esp寄存器的值。这里将ebp定义为指针，是为了方便后面取ebp寄存器的值。
* 调用read\_ebp函数来获取执行print\_stackframe函数时ebp寄存器的值，这里read\_ebp必须定义为inline函数，否则获取的是执行read\_ebp函数时的ebp寄存器的值。
* 调用read\_eip函数来获取当前指令的位置，也就是此时eip寄存器的值。这里read\_eip必须定义为常规函数而不是inline函数，因为这样的话在调用read\_eip时会把当前指令的下一条指令的地址（也就是eip寄存器的值）压栈，那么在进入read\_eip函数内部后便可以从栈中获取到调用前eip寄存器的值。
* 由于变量eip存放的是下一条指令的地址，因此将变量eip的值减去1，得到的指令地址就属于当前指令的范围了。由于只要输入的地址属于当前指令的起始和结束位置之间，print\_debuginfo都能搜索到当前指令，因此这里减去1即可。
* 以后变量eip的值就不能再调用read\_eip来获取了（每次调用获取的值都是相同的），而应该从ebp寄存器指向栈中的位置再往上一个单位中获取。
* 由于ebp寄存器指向栈中的位置存放的是调用者的ebp寄存器的值，据此可以继续顺藤摸瓜，不断回溯，直到ebp寄存器的值变为0

****

****

ebp:0x0007bf8 此时ebp的值是kern\_init函数的栈顶地址，从obj/bootblock.asm文件中知道整个栈的栈顶地址为0x00007c00，ebp指向的栈位置存放调用者的ebp寄存器的值，ebp+4指向的栈位置存放返回地址的值，这意味着kern\_init函数的调用者（也就是bootmain函数）没有传递任何输入参数给它！因为单是存放旧的ebp、返回地址已经占用8字节了。

eip:0x00007d6e eip的值是kern\_init函数的返回地址，也就是bootmain函数调用kern\_init对应的指令的下一条指令的地址。这与obj/bootblock.asm是相符合的。

args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8 一般来说，args存放的4个dword是对应4个输入参数的值。但这里比较特殊，由于bootmain函数调用kern\_init并没传递任何输入参数，并且栈顶的位置恰好在boot loader第一条指令存放的地址的上面，而args恰好是kern\_int的ebp寄存器指向的栈顶往上第2~5个单元，因此args存放的就是boot loader指令的前16个字节！可以对比obj/bootblock.asm文件来验证（验证时要注意系统是小端字节序）。

#### 练习六：完善中断初始化和处理

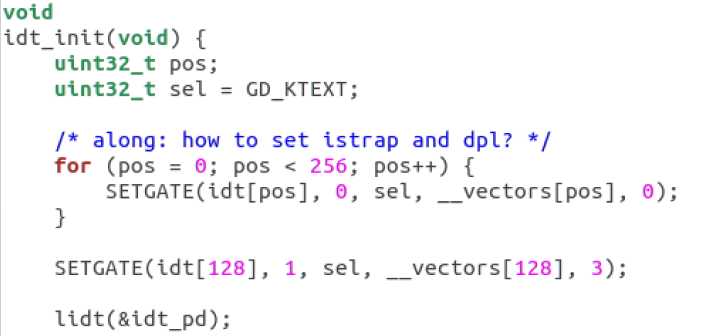
1.中断描述符表（也可简称为保护模式下的中断向量表）中一个表项占多少字节？其中哪几位代表中断处理代码的入口？

中断描述符表一个表项占8个字节，其结构如下：

* bit 63..48: offset 31..16
* bit 47..32: 属性信息，包括DPL、P flag等
* bit 31..16: Segment selector
* bit 15..0: offset 15..0

其中第16~32位是段选择子，用于索引全局描述符表GDT来获取中断处理代码对应的段地址，再加上第0~15、48~63位构成的偏移地址，即可得到中断处理代码的入口。

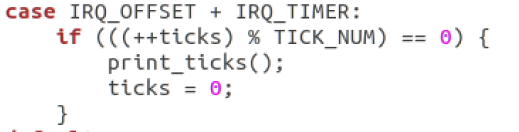
2.



idt\_init函数的功能是初始化IDT表。IDT表中每个元素均为门描述符，记录一个中断向量的属性，包括中断向量对应的中断处理函数的段选择子/偏移量、门类型、DPL等。因此，初始化IDT表实际上是初始化每个中断向量的这些属性。

1. 题目已经提供中断向量的门类型和DPL的设置方法：除了系统调用的门类型为陷阱门、DPL=3外，其他中断的门类型均为中断门、DPL均为0.
2. 中断处理函数的段选择子及偏移量的设置要参考kern/trap/vectors.S文件：由该文件可知，所有中断向量的中断处理函数地址均保存在\_\_vectors数组中，该数组中第i个元素对应第i个中断向量的中断处理函数地址。而且由文件开头可知，中断处理函数属于.text的内容。因此，中断处理函数的段选择子即.text的段选择子GD\_KTEXT。从kern/mm/pmm.c可知.text的段基址为0，因此中断处理函数地址的偏移量等于其地址本身。
3. 完成IDT表的初始化后，还要使用lidt命令将IDT表的起始地址加载到IDTR寄存器中。

3.



定义一个全局变量ticks，每次时钟中断将ticks加1，加到100后打印"100 ticks"，然后将ticks清零重新计数。

1. **实验总结**

通过本次实验，我学会了如何阅读源码以及对单个命令进行调试，对机器启动到操作系统运行的过程有了一个初步的了解。因为之前并没有接触过类似的实验，所以对于整个做实验的方式并不熟悉，之前学习的关于操作系统的知识也大多比较宽泛，没有实际的应用，对于很多源码的意义无法和知识结合起来，所以在做实验的时候查阅了很多资料以及询问同学。本次实验最大的收获就是了解了BIOS启动过程和bootloader启动过程，并且学会了如何调试命令执行情况，对于以后持续的学习有很大的帮助。

付子豪4042017006