调试分析 Linux 0.00 引导程序-实验报告

2021113117 王宇轩

1. `head.s` 的工作原理

当`boot.s`将`head.s`的代码移动到内存0开始处、设置好临时的`GDT`等数据结构、设置处理器运行在保护模式后，便通过`jmpi 0,8`指令跳转至head执行。

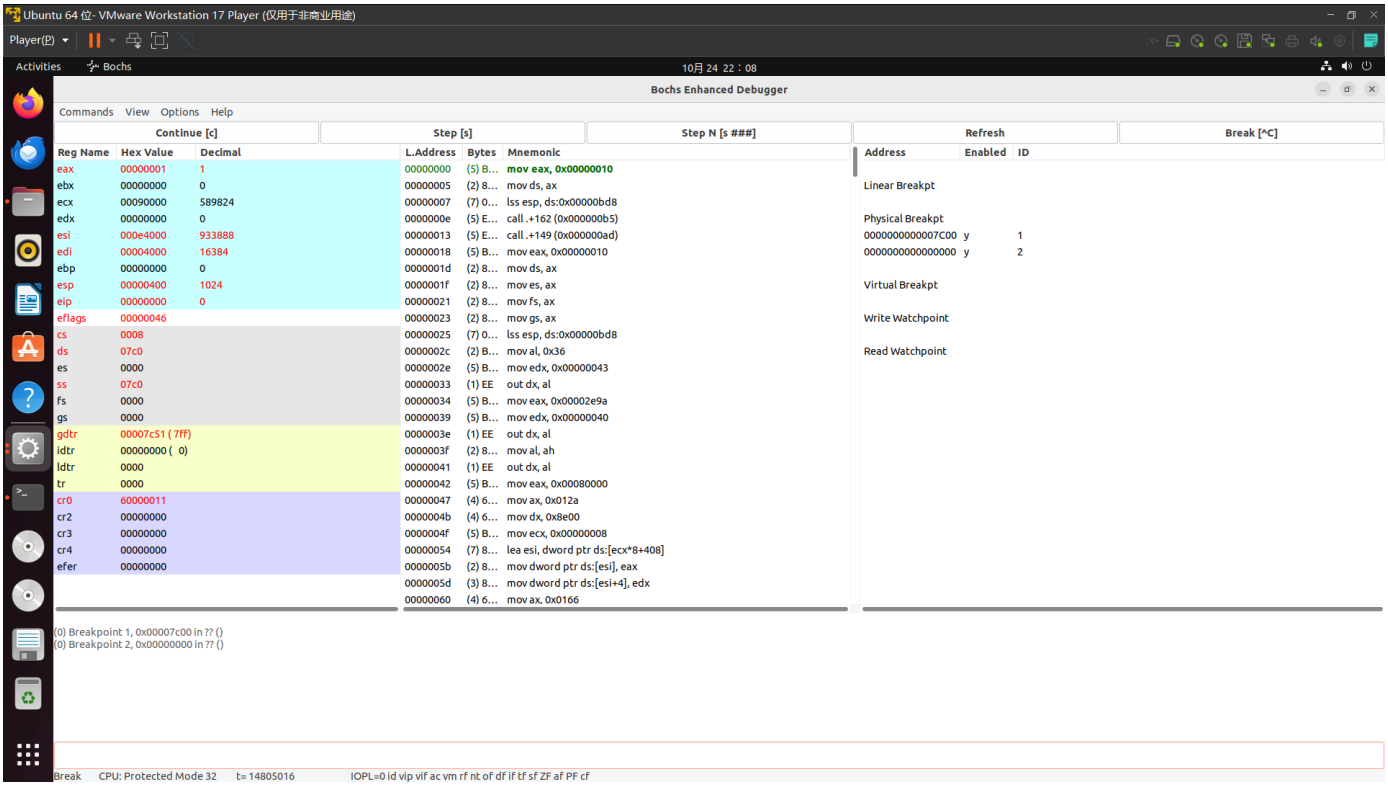
通过`bochs -q -f linux000_gui.bxrc`指令启动Linux 0.00的调试。其中，配置文件`linux000_gui.bxrc`的编写（主要参数）如下：

```
# 指明所模拟的PC机的ROM BIOS
romimage: file=$BXSHARE/BIOS-bochs-latest
# 设置被模拟系统所含内存容量
megs: 16
# 指明所模拟的PC机的VGA显示ROM程序
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
# floppya表示第一个软驱，floppyb代表第二个软驱。指明使用磁盘映像文件的名称。
floppya: 1_44="Image", status=inserted
# 用来定义模拟机器中用于引导启动的驱动器。可以指定是软盘、硬盘或者CDROM。也可以使用驱动器号'c'和'a'。
boot: a
# 指定bochs记录日志信息的路径名
log: bochsout.txt
# Linux下可视化配置
display_library: x, options="gui_debug"
```

开始调试后，使用指令`b 0x7c00`设置断点，`c`继续执行，此时断点就是`boot.s`代码起始位置。再次设置断点于`jmpi 0,8`指令处跟踪，此前程序已设置GDT如下图（可在bochs界面的view菜单中调出GDT查看）、通过`lmsw`指令设置运行于保护模式，此时`jmpi 0,8`的操作数`0,8`分别为**偏移量**和**段选择子**。段选择子8对应图中GDT第二条段描述符，对应着`head.s`的代码段，位于线性地址0处。

Index	Base Address	Size	DPL	Info
00 (Selector 0x0000)	0x0	0x0	0	Unused
01 (Selector 0x0008)	0x0	0x7FFFFF	0	32-bit code
02 (Selector 0x0010)	0x0	0x7FFFFF	0	32-bit data
03 (Selector 0x0018)	0x7000000	0xF0000FFF	0	
04 (Selector 0x0020)	0x0	0x7C51	0	
05 (Selector 0x0028)	0x0	0x0	0	
06 (Selector 0x0030)	0x0	0x0	0	
07 (Selector 0x0038)	0x0	0x0	0	

通过跳转指令跳转至`head.s`执行，此时通用寄存器值显示在窗口左侧，使用命令`info ldt`等，就可查看ldt等系统数据结构的值。如果要查看某物理地址的内容，如`0x7c00`，可用指令`xp 0x7c00`。



`head.s`程序运行在32位保护模式下，其中主要包括初始设置的代码、时钟中断int0x08的过程代码、系统调用中断it0x80的过程代码以及任务A和任务B等的代码和数据。其中初始设置工作主要包括：**①重新设置GDT表；②设置系统定时器芯片；③重新设置IDT表并且设置时钟和系统调用中断门；④移动到任务A中执行。**

1.1 设置GDT和IDT

首先加载数据段寄存器DS、堆栈段寄存器SS和堆栈指针ESP。所有段的线性基地址都是0。在新的位置重新设置IDT和GDT表。

```
startup_32:
    movl $0x10,%eax
    mov %ax,%ds
#   mov %ax,%es
    lss init_stack,%esp

# setup base fields of descriptors.
    call setup_idt
    call setup_gdt
    movl $0x10,%eax     # reload all the segment registers
    mov %ax,%ds     # after changing gdt.
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
    lss init_stack,%esp
```

设置后的GDT如下。

Index	Base Address	Size	DPL	Info
00 (Selector 0x0000)	0x0	0x0	0	Unused
01 (Selector 0x0008)	0x0	0x7FFFFF	0	32-bit code
02 (Selector 0x0010)	0x0	0x7FFFFF	0	32-bit data
03 (Selector 0x0018)	0xB8000	0x2FFF	0	32-bit data
04 (Selector 0x0020)	0xBF8	0x68	3	Available 32bit TSS
05 (Selector 0x0028)	0xBE0	0x40	3	LDT
06 (Selector 0x0030)	0xE78	0x68	3	Available 32bit TSS
07 (Selector 0x0038)	0xE60	0x40	3	LDT

代码`setup_idt`暂时设置IDT表中所有256个中断门描述符都为同一个默认值，均使用默认的中断处理过程`ignore int.`。设置的具体方法是：首先在eax和edx寄存器对中分别设置好默认中断门描述符的0-3字节和4-7字节的内容，然后利用该寄存器对循环往IDT表中填充默认中断门描述符内容。设置后的IDT如下。

Interrupt	L.Address
00	0x0008:0x00000114
01	0x0008:0x00000114
02	0x0008:0x00000114
03	0x0008:0x00000114
04	0x0008:0x00000114
05	0x0008:0x00000114
06	0x0008:0x00000114
07	0x0008:0x00000114
08	0x0008:0x00000114
09	0x0008:0x00000114

1.2 设置系统定时器芯片

通过调用I/O实现。

```
# 设置8253定时芯片。把计数器通道0设置成每隔10毫秒向中断控制器发送一个中断请求信号。
   movb $0x36,%al      # 控制字：设置通道0工作在方式3、计数初值采用二进制。
   mov1 $0x43,%edx     # 8253芯片控制字寄存器写端口。
   outb %al,%dx
   mov1 $LATCH,%eax    # 初始计数值设置为LATCH(1193180/100),即频率100HZ。
   movl $0x40,%edx     # 通道0的端口。
   outb %al,%dx        # 分两次把初始计数值写入通道0。
   movb %ah,%al
   outb %al,%dx
```

1.3 重新设置IDT表并且设置时钟和系统调用中断门

在**1.1 设置GDT和IDT**中，程序完成了对IDT的初始化，接下来程序用以下代码在IDT表第8和第128（0x80）项处分别设置定时中断门描述符和系统调用陷阱门描述符：

```
movl $0x00080000,%eax        #中断程序属内核，即EAX高字是内核代码段选择符0x0008。
movw $timer_interrupt,%ax    #设置定时中断门描述符。取定时中断处理程序地址。
movw $0x8E00,%dx             #中断门类型是14（屏蔽中断），特权级0或硬件使用。
movl $0x08,%ecx              #开机时BIOS设置的时钟中断向量号8。这里直接使用它。
lea idt(,%ecx,8),%esi        #把IDT描述符0x08地址放入ESI中，然后设置该描述符。
movl %eax,(%esi)
movl %edx,4(%esi)
movw $system_interrupt,%ax   #设置系统调用陷阱门描述符。取系统调用处理程序地址。
movw $Oxef00,%dx             #陷阱门类型是15，特权级3的程序可执行。
movl $0x80,%ecx              #系统调用向量号是0x80。
lea idt (%ecx,8),%esi        #把IDT描述符项0x80地址放入ESI中，然后设置该描述符。
movl %eax,(%esi)
movl %edx,4(%esi)
```

设置好的IDT如下，除了0x8和0x80处的描述符被修改，其它没有变化。

```
07      0x0008:0x00000114
08      0x0008:0x0000012A
09      0x0008:0x00000114
80      0x0008:0x00000166
```

1.4 移动到任务A中执行

`head.s`是通过**人工建立好中断返回时的堆栈场景后，使用`iret`指令，虚拟地"从中断返回"至任务0执行的。**代码如下：

```
#好了，现在我们为移动到任务0（任务A)中执行来操作堆栈内容，在堆栈中人工建立中断返回时的场景。
  pushfl                     #复位标志寄存器EFLAGS中的嵌套任务标志。
  andl $Oxffffbfff,(%esp)
  popfl
  mov1 STSSO SEL,%eax        #把任务0的TSS段选择符加载到任务寄存器TR。
  ltr %ax
  mov1 $LDTO SEL,%eax        #把任务0的LDT段选择符加载到局部描述符表寄存器LDTR。
  lldt %ax                   #TR和LDTR只需人工加载一次，以后CPU会自动处理。
  movl $0,current            #把当前任务号0保存在current变量中。
  sti                        #现在开启中断，并在栈中营造中断返回时的场景。
  pushl $0x17                #把任务0当前局部空间数据段（堆栈段）选择符入栈。
  pushl $init stack          #把堆栈指针入栈（也可以直接把SP入栈）。
  pushfl                     #把标志寄存器值入栈。
  pushl $0xOf                #把当前局部空间代码段选择符入栈。
  pushl $task0               #把代码指针入栈。
  iret                       #执行中断返回指令，从而切换到特权级3的任务0中执行。
```

在后面的报告中会对这部分代码的执行作出更为详细的分析。

2. `head.s` 的内存分布状况

各部分的起始、终止地址可分别用如下方式找到：

- 在已设置好的GDT（见**1.1 设置GDT和IDT**中图片）中，可以找到两个任务LDT和TSS的起始地址。
- 在进入任务执行后，通过查看寄存器的值也可看到当前任务的LDT、TSS地址（LDTR、TR寄存器值），如下图，为任务0执行时的寄存器情况。

- 上图中也可以查看段寄存器中保存的段选择子，对应着LDT中的段描述符。用 `info ldt` 指令即可查看LDT，任务0的LDT如下。例如，上图的CS中代码段段选择子中索引（高13位）为0x01，即可在LDT中找到对应的段描述符，从而获得地址。

```
Local Descriptor Table (base=0x00000be0, limit=64):
LDT[0x00]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x01]=Code segment, base=0x00000000, limit=0x003fffff, Execute/Read, Non-Conforming,
Accessed, 32-bit
LDT[0x02]=Data segment, base=0x00000000, limit=0x003fffff, Read/Write, Accessed
LDT[0x03]=32-Bit Interrupt Gate target=0x0000:0x00000000, DPL=0
LDT[0x04]=??? descriptor hi=0x00000000, lo=0x00000010
LDT[0x05]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x06]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x07]=??? descriptor hi=0x00000000, lo=0x00000000
You can list individual entries with 'info ldt [NUM]' or groups with 'info ldt [NUM]
[NUM]'
```

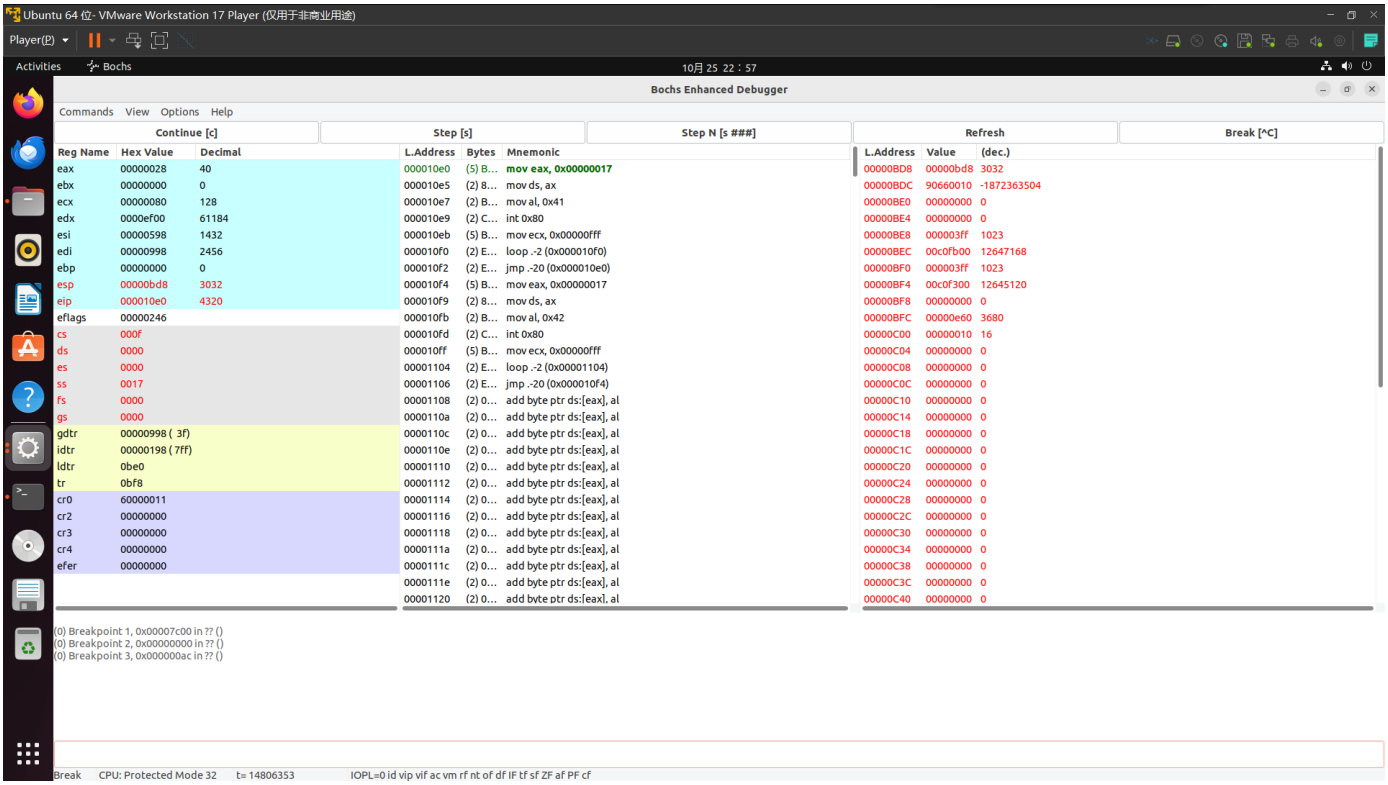
此外，通过分析 `head.s` 中各个段描述符定义也可得到地址。

实际上，本内核示例中所有代码和数据段都对应到物理内存同一个区域上，即从物理内存0开始的区域。GDT中全局代码段和数据段描述符的内容都设置为：基地址为0x0000；段限长值为0x07ff。因为颗粒度为1，所以实际段长度为 8MB。而全局显示数据段被设置成：基地址为0xb8000；段限长值为0x0002，所以实际段长度为8KB，对应到显示内存区域上。两个任务的在LDT中代码段和数据段描述符的内容也都设置为：基地址为0x0000:段限长值为0x03ff,实际段长度为4MB。因此在线性地址空间中这个“内核”的代码和数据段与任务的代码和数据段都从线性地址0开始并且由于没有采用分页机制，所以它们都直接对应物理地址0开始处。

至于任务的堆栈段，在TSS中有定义内核栈（ss:exp(0)）和用户栈（ss:esp），如tss1，在调试中任务一执行时，可通过命令 `info tss` 查看当前任务的TSS，输出结果如下。

```
tr:s=0x30, base=0x00000e78, valid=1
ss:esp(0): 0x0010:0x000010e0
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x000010f4
eflags: 0x00000200
cs: 0x000f ds: 0x0017 ss: 0x0017
es: 0x0017 fs: 0x0017 gs: 0x0017
eax: 0x00000000  ebx: 0x00000000  ecx: 0x00000000  edx: 0x00000000
esi: 0x00000000  edi: 0x00000000  ebp: 0x00000000  esp: 0x00001308
ldt: 0x0038
i/o map: 0x0800
```

任务0的用户栈即初始栈，其esp寄存器已设定，因此没有在TSS中体现。在Bochs中，可调出View-Stack查看用户栈，如图。



综上，可以得到**每个数据段，代码段，栈段的起始与终止的内存地址如下：**

1. 内核： 代码段：0x0~0x07ffff 数据段：0x0~0x07ffff
2. 全局显示数据段：0xb8000~0x000bafff
3. 任务0： 代码段：0x0~0x003fffff 数据段：0x0~0x003fffff 内核栈：0x00000e60~0x00000c61 用户栈（即初始栈）：0x00000bd8~0x000009d9

4. 任务1： 代码段：0x0~0x003fffff 数据段：0x0~0x003fffff 内核栈：0x000010e0~0x00000ee1 用户栈：0x00001308~0x00001109

3. `head.s` 57至62行

由于处于特权级0的代码不能直接把控制权转移到特权级3的代码中执行，但中断返回操作是可以的，因此当初始化GDT、IDT和定时芯片结束后，我们就利用中断返回指令IRET来启动运行第1个任务。具体实现方法是在初始堆栈init_stack中人工设置一个返回环境。即把任务0的TSS段选择符加载到任务寄存器LTR中、LDT段选择符加载到LDTR中以后，把任务0的用户栈指针(0x17:init stack)和代码指针(0x0f:task0)以及标志寄存器值压入栈中，然后执行中断返回指令IRET。该指令会弹出堆栈上的堆栈指针作为任务0用户栈指针，恢复假设的任务0的标志寄存器内容，并且弹出栈中代码指针放入CS:EIP寄存器中，从而开始执行任务0的代码，完成了从特权级0到特权级3代码的控制转移。

4. `iret` 执行后， `pc` 如何找到下一条指令

从处理器的视角看，执行 `iret` 指令意味着要从中断返回至之前的程序中运行，此时处理器检查EFLAGS中的NT标志位，发现为0，即没有任务嵌套，于是IRET指令只会从中断堆栈中弹出EIP值作为 `pc` 指针的值，继续执行先前的代码。

在bochs中调试此程序即将执行62行的 `iret` 时，可用 `info eflags` 指令查看标志状态，输出如下，其中 `nt` 为小写，代表NT标志位为0。

```
id vip vif ac vm rf nt IOPL=0 of df IF tf sf ZF af PF cf
```

此时的栈如下图，栈顶所存的0x10e0就是任务0第一条指令的地址。

L.Address	Value	(dec.)
00000BC4	000010e0	4320
00000BC8	0000000f	15
00000BCC	00000246	582
00000BD0	00000bd8	3032
00000BD4	00000017	23
00000BD8	00000bd8	3032
00000BDC	90660010	-1872363504
00000BE0	00000000	0
00000BE4	00000000	0
00000BE8	000003ff	1023
00000BEC	00c0fa00	12646912
00000BF0	000003ff	1023
00000BF4	00c0f200	12644864
00000BF8	00000000	0
00000BFC	00000e60	3680
00000C00	00000010	16
00000C04	00000000	0

在我们的视角来看，其实是程序的57至62行设置好了任务0的运行环境，并手动将EIP等信息压入堆栈，使得`iret`执行后可以开始执行任务0。

5. `iret`执行前后，栈的变化

上图即为执行前的栈。**用指令`s`（或可视化界面的step按键）进行单步运行**，可查看执行后的栈如下。

L.Address	Value	(dec.)
00000BD8	00000bd8	3032
00000BDC	90660010	-1872363504
00000BE0	00000000	0
00000BE4	00000000	0
00000BE8	000003ff	1023
00000BEC	00c0fb00	12647168
00000BF0	000003ff	1023
00000BF4	00c0f300	12645120
00000BF8	00000000	0
00000BFC	00000e60	3680
00000C00	00000010	16

可见， 此前压入栈的（从下至上：）任务0当前局部空间数据段（堆栈段）选择符、堆栈指针、标志寄存器值、当前局部空间代码段选择符、代码指针被弹出。

6. 任务进行系统调用时栈的变化情况

上图即为任务调用`int 0x80`前栈的情况。调用后，栈如下：

L.Address	Value	(dec.)
00000E4C	000010eb	4331
00000E50	0000000f	15
00000E54	00000246	582
00000E58	00000bd8	3032
00000E5C	00000017	23
00000E60	00000000	0
00000E64	00000000	0
00000E68	000003ff	1023
00000E6C	00c0fa00	12646912

此时已切换至任务0的内核栈，从0x00000e60地址处开始，又向栈内压入了为任务0保存的原来的用户栈栈顶地址0x17:0x0BD8、标志寄存器值和中断返回地址0x0F:0x10EB。

7. 对剩余问题的回答

实验指导书的**实验内容**中的大多数问题已在上述实验报告中回答，这里对其余问题进行回答。

7.1 如何在内存指定地方进行反汇编?

可用指令`u/行数 开始地址`，如想要反汇编`0x7c00`处开始的5条代码，即可用指令`u/5 0x7c00`。若不提供地址则从当前地址开始反汇编。

7.2 如何把真正的内核程序从硬盘或软驱装载到自己想要放的地方?

boot.s程序会首先利用R0MBI0S中断int0x13把软盘中的head代码读入到内存0x10000(64KB)位置开始处，然后再把这段head代码移动到内存0开始处。

```
!加载内核代码到内存0x10000开始处。
load_system:
        mov dx,#0x0000          !利用BIOS中断int0x13功能2从启动盘读取head代码。
        mov cX,#0x0002          !DH-磁头号；DL-驱动器号；CH-10位磁道号低8位；
        mov ax,#SYSSEG          !CL-位7、6是磁道号高2位，位5-0起始扇区号（从1计）。
        mov es,ax               !ES:BX-读入缓冲区位置(0x1000:0x0000)。
        xor bx,bx               !AH-读扇区功能号；AL-需读的扇区数(17)。
        mov ax,#0x200+SYSLEN
        int 0x13
        jnc ok_load             !若没有发生错误则跳转继续运行，否则死循环。
die:    Jmp die                 !把内核代码移动到内存0开始处。共移动8KB字节（内核长度不超过
8KB)。
ok_load:
        cli                     !关中断。
```

```
    mov ax,#SYSSEG          !移动开始位置DS:SI=0x1000:0;目的位置ES:DI=0:0。
    mov ds,ax
    xor ax,ax
    mov es,ax
    mov cx,#0x1000          !设置共移动4K次，每次移动一个字(word)。
    sub si,si
    sub di,di
    rep movw                !执行重复移动指令。
```

不能让boot程序把head代码从软盘或映像文件中直接加载到内存0处。因为加载操作需要使用ROM BIOS提供的中断过程，而BIOS使用的中断向量表正处于内存0开始的地方，并且在内存1KB开始处是BIOS程序使用的数据区，所以若直接把head代码加载到内存0处将使得BIOS中断过程不能正常运行。

7.3 如何查看实模式的中断程序?

调出IDT，查中断向量对应的描述符即可获得对应中断程序的起始地址，然后用`xp`指令查看对应地址的内容即可查看实模式的中断程序。

7.4 如何静态创建 gdt 与 idt ?

`head.s`是通过`setup_gdt`和`setup_idt`进行的，即在程序中写好描述符表后，将对应地址装在入GDTR和IDTR中。

7.5 如何从实模式切换到保护模式?

对应`boot.s`中这段代码：

```
!设置控制寄存器CR0(即机器状态字)，进入保护模式。段选择符值8对应GDT表中第2个段描述符。
  mov ax,#0x0001    !在CR0中设置保护模式标志PE(位0)。
  lmsw ax           !然后跳转至段选择符值指定的段中，偏移0处。
  jmpi 0,8          !注意此时段值已是段选择符。该段的线性基地址是0。
```

7.6 调试跟踪`jmpi 0,8`，解释如何寻址?

已在第1部分回答。