




第3章 引导启动程序（boot）

3.1 概述

本章主要描述 boot/目录中的三个汇编代码文件，见列表 3-1 所示。正如在前一章中提到的，这三个文件虽然都是汇编程序，但却使用了两种语法格式。bootsect.s 和 setup.s 采用近似于 Intel 的汇编语言语法，需要使用 Intel 8086 汇编编译器和连接器 as86 和 ld86，而 head.s 则使用 GNU 的汇编程序格式，并且运行在保护模式下，需要用 GNU 的 as 进行编译。这是一种 AT&T 语法的汇编语言程序。

使用两种编译器的主要原因是由于对于 Intel x86 处理器系列来讲，GNU 的编译器仅支持 i386 及以后出的 CPU。不支持生成运行在实模式下的程序。

列表 3-1 linux/boot/目录

	文件名	长度(字节)	最后修改时间(GMT)	说明
	bootsect.s	5052 bytes	1991-12-05 22:47:58	
	head.s	5938 bytes	1991-11-18 15:05:09	
	setup.s	5364 bytes	1991-12-05 22:48:10	

阅读这些代码除了你需要知道一些一般 8086 汇编语言的知识以外，还要对采用 Intel 80X86 微处理器的 PC 机的体系结构以及 80386 32 位保护模式下的编程原理有些了解。所以在开始阅读源代码之前可以先大概浏览一下附录中有关 PC 机硬件接口控制编程和 80386 32 位保护模式的编程方法，在阅读代码时再就事论事地针对具体问题参考附录中的详细说明。

3.2 总体功能

这里先总的说明一下 Linux 操作系统启动部分的主要执行流程。当 PC 的电源打开后，80x86 结构的 CPU 将自动进入实模式，并从地址 0xFFFF0 开始自动执行程序代码，这个地址通常是 ROM-BIOS 中的地址。PC 机的 BIOS 将执行某些系统的检测，并在物理地址 0 处开始初始化中断向量。此后，它将可启动设备的第一个扇区（磁盘引导扇区，512 字节）读入内存绝对地址 0x7C00 处，并跳转到这个地方。启动设备通常是软驱或是硬盘。这里的叙述是非常简单的，但这已经足够理解内核初始化的工作过程了。

Linux 的最最前面部分是用 8086 汇编语言编写的(boot/bootsect.s)，它将由 BIOS 读入到内存绝对地址 0x7C00(31KB)处，当它被执行时就会把自己移到绝对地址 0x90000(576KB)处，并把启动设备中后 2kB 字节代码(boot/setup.s)读入到内存 0x90200 处，而内核的其它部分（system 模块）则被读入到从地址 0x10000 开始处，因为当时 system 模块的长度不会超过 0x80000 字节大小（即 512KB），所以它不会覆盖在 0x90000 处开始的 bootsect 和 setup 模块。后面 setup 程序将会把 system 模块移动到内存起始处，这样 system 模块中代码的地址也即等于实际的物理地址，便于对内核代码和数据的操作。图 3-1 清晰地显示出 Linux 系统启动时这几个程序或模块在内存中的动态位置。其中，每一竖条框代表某一时刻内存中

各程序的映像位置图。在系统加载期间将显示信息"Loading..."。然后控制权将传递给 boot/setup.s 中的代码，这是另一个实模式汇编语言程序。

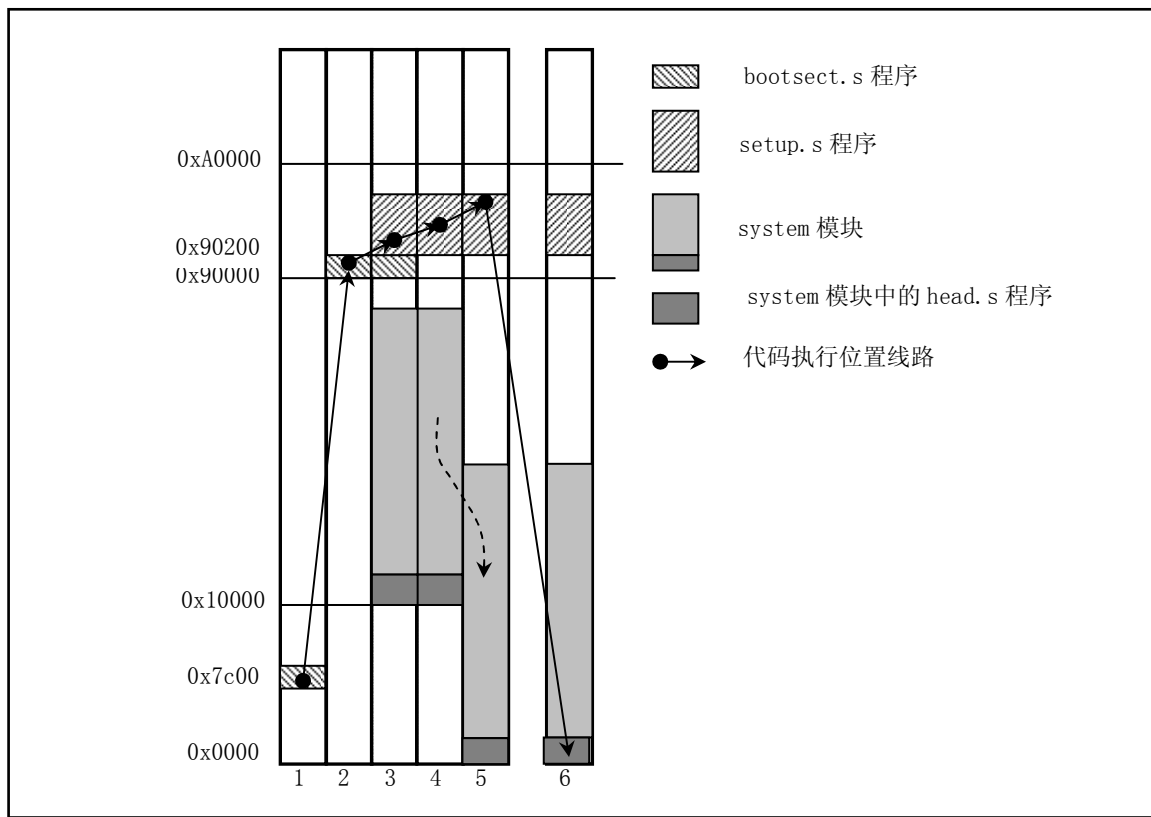


图 3-1 启动引导时内核在内存中的位置和移动后的位置情况

启动部分识别主机的某些特性以及 vga 卡的类型。如果需要，它会要求用户为控制台选择显示模式。然后将整个系统从地址 0x10000 移至 0x0000 处，进入保护模式并跳转至系统的余下部分（在 0x0000 处）。此时所有 32 位运行方式的设置启动被完成：IDT、GDT 以及 LDT 被加载，处理器和协处理器也已确认，分页工作也设置好了；最终调用 init/main.c 中的 main() 程序。上述操作的源代码是在 boot/head.S 中的，这可能是整个内核中最有诀窍的代码了。注意如果在前述任何一步中出了错，计算机就会死锁。在操作系统还没有完全运转之前是处理不了出错的。

为什么不把系统模块直接加载到物理地址 0x0000 开始处而要在 setup 程序中再进行移动呢？这是因为在 setup 程序代码开始部分还需要利用 ROM BIOS 中的中断调用来获取机器的一些参数（例如显卡模式、硬盘参数表等）。当 BIOS 初始化时会在物理内存开始处放置一个大小为 0x400 字节(1Kb)的中断向量表，因此需要在使用完 BIOS 的中断调用后才能将这个区域覆盖掉。

3.3 bootsect.s 程序

3.3.1 功能描述

bootsect.s 代码是磁盘引导块程序，驻留在磁盘的第一个扇区中（引导扇区，0 磁道（柱面），0 磁头，第 1 个扇区）。在 PC 机加电 ROM BIOS 自检后，引导扇区由 BIOS 加载到内存 0x7c00 处，然后将自己移动到内存 0x90000 处。该程序的主要作用是首先将 setup 模块（由 setup.s 编译成）从磁盘加载到内存，紧接着 bootsect 的后面位置（0x90200），然后利用 BIOS 中断 0x13 取磁盘参数表中当前启动引导盘的参

数，接着在屏幕上显示“Loading system...”字符串。再者将 system 模块从磁盘上加载到内存 0x10000 开始的地方。随后确定根文件系统的设备号，若没有指定，则根据所保存的引导盘的每磁道扇区数判别出盘的类型和种类（是 1.44M A 盘吗？）并保存其设备号于 root_dev(引导块的 0x508 地址处)，最后长跳转到 setup 程序的开始处（0x90200）执行 setup 程序。

3.3.2 代码注释

程序 3-1 linux/boot/bootsect.s

```

1 !
2 ! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
3 ! 0x3000 is 0x30000 bytes = 196kB, more than enough for current
4 ! versions of linux
5 ! SYS_SIZE 是要加载的节数（16 字节为 1 节）。0x3000 共为
6 ! 0x30000 字节=196 kB（若以 1024 字节为 1KB 计，则应该是 192KB），对于当前的版本空间已足够了。
7 !
8 SYSSIZE = 0x3000      ! 指编译连接后 system 模块的大小。参见列表 2.1 中第 92 的说明。
9 !                    ! 这里给出了一个最大默认值。
10 !
11 bootsect.s          (C) 1991 Linus Torvalds
12 !
13 ! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
14 ! itself out of the way to address 0x90000, and jumps there.
15 !
16 ! It then loads 'setup' directly after itself (0x90200), and the system
17 ! at 0x10000, using BIOS interrupts.
18 !
19 ! NOTE! currently system is at most 8*65536 bytes long. This should be no
20 ! problem, even in the future. I want to keep it simple. This 512 kB
21 ! kernel size should be enough, especially as this doesn't contain the
22 ! buffer cache as in minix
23 !
24 ! The loader has been made as simple as possible, and continuous
25 ! read errors will result in a unbreakable loop. Reboot by hand. It
26 ! loads pretty fast by getting whole sectors at a time whenever possible.
27 !
28 ! 以下是前面这些文字的翻译：
29 ! bootsect.s          (C) 1991 Linus Torvalds 版权所有
30 !
31 ! bootsect.s 被 bios-启动子程序加载至 0x7c00 (31k)处，并将自己
32 ! 移到了地址 0x90000 (576k)处，并跳转至那里。
33 !
34 ! 它然后使用 BIOS 中断将 'setup' 直接加载到自己的后面(0x90200) (576.5k)，
35 ! 并将 system 加载到地址 0x10000 处。
36 !
37 ! 注意！目前的内核系统最大长度限制为(8*65536) (512k) 字节，即使是在
38 ! 将来这也应该没有问题的。我想让它保持简单明了。这样 512k 的最大内核长度应该
39 ! 足够了，尤其是这里没有象 minix 中一样包含缓冲区高速缓冲。
40 !
41 ! 加载程序已经做的够简单了，所以持续的读出错将导致死循环。只能手工重启。
42 ! 只要可能，通过一次读取所有的扇区，加载过程可以做的很快。

```

24

```

25 .globl begtext, begdata, begbss, endtext, enddata, endbss ! 定义了6个全局标识符;
26 .text ! 文本段;
27 begtext:
28 .data ! 数据段;
29 begdata:
30 .bss ! 未初始化数据段(Block Started by Symbol);
31 begbss:
32 .text ! 文本段;
33
34 SETUPLEN = 4 ! nr of setup-sectors
! setup 程序的扇区数(setup-sectors)值;
35 BOOTSEG = 0x07c0 ! original address of boot-sector
! bootsect 的原始地址(是段地址, 以下同);
36 INITSEG = 0x9000 ! we move boot here - out of the way
! 将 bootsect 移到这里 -- 避开;
37 SETUPSEG = 0x9020 ! setup starts here
! setup 程序从这里开始;
38 SYSSEG = 0x1000 ! system loaded at 0x10000 (65536).
! system 模块加载到 0x10000 (64 kB) 处;
39 ENDSEG = SYSSEG + SYSSIZE ! where to stop loading
! 停止加载的段地址;
40
41 ! ROOT_DEV: 0x000 - same type of floppy as boot.
! 根文件系统设备使用与引导时同样的软驱设备;
42 ! 0x301 - first partition on first drive etc
! 根文件系统设备在第一个硬盘的第一个分区上, 等等;
43 ROOT_DEV = 0x306 ! 指定根文件系统设备是第2个硬盘的第1个分区。这是Linux老式的硬盘命名
! 方式, 具体值的含义如下:
! 设备号=主设备号*256 + 次设备号(也即 dev_no = (major<<8) + minor)
! (主设备号: 1-内存, 2-磁盘, 3-硬盘, 4-ttyx, 5-tty, 6-并行口, 7-非命名管道)
! 0x300 - /dev/hd0 - 代表整个第1个硬盘;
! 0x301 - /dev/hd1 - 第1个盘的第1个分区;
! ...
! 0x304 - /dev/hd4 - 第1个盘的第4个分区;
! 0x305 - /dev/hd5 - 代表整个第2个硬盘;
! 0x306 - /dev/hd6 - 第2个盘的第1个分区;
! ...
! 0x309 - /dev/hd9 - 第2个盘的第4个分区;
! 从linux内核0.95版后已经使用与现在相同的命名方法了。
44
45 entry start ! 告知连接程序, 程序从 start 标号开始执行。
46 start: ! 47--56 行作用是将自身(bootsect)从目前段位置 0x07c0(31k)
! 移动到 0x9000(576k)处, 共 256 字(512 字节), 然后跳转到
! 移动后代码的 go 标号处, 也即本程序的下一语句处。
47 mov ax, #BOOTSEG ! 将 ds 段寄存器置为 0x7C0;
48 mov ds, ax
49 mov ax, #INITSEG ! 将 es 段寄存器置为 0x9000;
50 mov es, ax
51 mov cx, #256 ! 移动计数值=256 字;
52 sub si, si ! 源地址 ds:si = 0x07C0:0x0000
53 sub di, di ! 目的地址 es:di = 0x9000:0x0000
54 rep ! 重复执行, 直到 cx = 0
55 movw ! 移动 1 个字;

```

```

56      jmp     go, INITSEG      ! 间接跳转。这里 INITSEG 指出跳转到的段地址。
                                   ! 从下面开始, CPU 执行已移动到 0x9000 段处的代码。
57 go:    mov     ax, cs        ! 将 ds、es 和 ss 都置成移动后代码所在的段处 (0x9000)。
58      mov     ds, ax        ! 由于程序中有堆栈操作 (push, pop, call), 因此必须设置堆栈。
59      mov     es, ax
60 ! put stack at 0x9ff00.      ! 将堆栈指针 sp 指向 0x9ff00 (即 0x9000:0xff00) 处
61      mov     ss, ax
62      mov     sp, #0xFF00    ! arbitrary value >>512
                                   ! 由于代码段移动过了, 所以要重新设置堆栈段的位置。
                                   ! sp 只要指向远大于 512 偏移 (即地址 0x90200) 处
                                   ! 都可以。因为从 0x90200 地址开始处还要放置 setup 程序,
                                   ! 而此时 setup 程序大约为 4 个扇区, 因此 sp 要指向大
                                   ! 于 (0x200 + 0x200 * 4 + 堆栈大小) 处。

63
64 ! load the setup-sectors directly after the bootblock.
65 ! Note that 'es' is already set up.
   ! 在 bootsect 程序块后紧跟着加载 setup 模块的代码数据。
   ! 注意 es 已经设置好了。(在移动代码时 es 已经指向目的段地址处 0x9000)。

66
67 load_setup:
   ! 68--77 行的用途是利用 BIOS 中断 INT 0x13 将 setup 模块从磁盘第 2 个扇区
   ! 开始读到 0x90200 开始处, 共读 4 个扇区。如果读出错, 则复位驱动器, 并
   ! 重试, 没有退路。INT 0x13 的使用方法如下:
   ! 读扇区:
   ! ah = 0x02 - 读磁盘扇区到内存; al = 需要读出的扇区数量;
   ! ch = 磁道(柱面)号的低 8 位;    cl = 开始扇区 (0-5 位), 磁道号高 2 位 (6-7);
   ! dh = 磁头号;                    dl = 驱动器号 (如果是硬盘则位 7 要置位);
   ! es:bx → 指向数据缓冲区; 如果出错则 CF 标志置位。
68      mov     dx, #0x0000      ! drive 0, head 0
69      mov     cx, #0x0002      ! sector 2, track 0
70      mov     bx, #0x0200      ! address = 512, in INITSEG
71      mov     ax, #0x0200+SETUPLEN ! service 2, nr of sectors
72      int     0x13             ! read it
73      jnc     ok_load_setup    ! ok - continue
74      mov     dx, #0x0000
75      mov     ax, #0x0000      ! reset the diskette
76      int     0x13
77      j       load_setup
78
79 ok_load_setup:
80
81 ! Get disk drive parameters, specifically nr of sectors/track
   ! 取磁盘驱动器的参数, 特别是每道的扇区数量。
   ! 取磁盘驱动器参数 INT 0x13 调用格式和返回信息如下:
   ! ah = 0x08    dl = 驱动器号 (如果是硬盘则要置位 7 为 1)。
   ! 返回信息:
   ! 如果出错则 CF 置位, 并且 ah = 状态码。
   ! ah = 0,    al = 0,          bl = 驱动器类型 (AT/PS2)
   ! ch = 最大磁道号的低 8 位, cl = 每磁道最大扇区数 (位 0-5), 最大磁道号高 2 位 (位 6-7)
   ! dh = 最大磁头号,          dl = 驱动器数量,
   ! es:di → 软驱磁盘参数表。
82
83      mov     dl, #0x00

```

```

84      mov     ax, #0x0800          ! AH=8 is get drive parameters
85      int     0x13
86      mov     ch, #0x00
87      seg cs          ! 表示下一条语句的操作数在 cs 段寄存器所指的段中。
88      mov     sectors, cx        ! 保存每磁道扇区数。
89      mov     ax, #INITSEG
90      mov     es, ax            ! 因为上面取磁盘参数中断改掉了 es 的值，这里重新改回。
91
92 ! Print some inane message      ! 显示一些信息('Loading system ...' 回车换行，共 24 个字符)。
93
94      mov     ah, #0x03          ! read cursor pos
95      xor     bh, bh            ! 读光标位置。
96      int     0x10
97
98      mov     cx, #24            ! 共 24 个字符。
99      mov     bx, #0x0007        ! page 0, attribute 7 (normal)
100     mov     bp, #msg1          ! 指向要显示的字符串。
101     mov     ax, #0x1301        ! write string, move cursor
102     int     0x10              ! 写字符串并移动光标。
103
104 ! ok, we've written the message, now
105 ! we want to load the system (at 0x10000) ! 现在开始将 system 模块加载到 0x10000 (64k) 处。
106
107     mov     ax, #SYSSEG
108     mov     es, ax            ! segment of 0x010000 ! es = 存放 system 的段地址。
109     call    read_it          ! 读磁盘上 system 模块，es 为输入参数。
110     call    kill_motor      ! 关闭驱动器马达，这样就可以知道驱动器的状态了。
111
112 ! After that we check which root-device to use. If the device is
113 ! defined (!= 0), nothing is done and the given device is used.
114 ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
115 ! on the number of sectors that the BIOS reports currently.
! 此后，我们检查要使用哪个根文件系统设备（简称根设备）。如果已经指定了设备(!=0)
! 就直接使用给定的设备。否则就需要根据 BIOS 报告的每磁道扇区数来
! 确定到底使用/dev/PS0 (2,28) 还是 /dev/at0 (2,8)。
! 上面一行中两个设备文件的含义：
! 在 Linux 中软驱的主设备号是 2(参见第 43 行的注释)，次设备号 = type*4 + nr，其中
! nr 为 0-3 分别对应软驱 A、B、C 或 D；type 是软驱的类型 (2→1.2M 或 7→1.44M 等)。
! 因为 7*4 + 0 = 28，所以 /dev/PS0 (2,28)指的是 1.44M A 驱动器,其设备号是 0x021c
! 同理 /dev/at0 (2,8)指的是 1.2M A 驱动器，其设备号是 0x0208。
116
117     seg cs
118     mov     ax, root_dev      ! 将根设备号
119     cmp     ax, #0
120     jne     root_defined
121     seg cs
122     mov     bx, sectors      ! 取上面第 88 行保存的每磁道扇区数。如果 sectors=15
! 则说明是 1.2Mb 的驱动器；如果 sectors=18，则说明是
! 1.44Mb 软驱。因为是可引导的驱动器，所以肯定是 A 驱。
123     mov     ax, #0x0208      ! /dev/ps0 - 1.2Mb
124     cmp     bx, #15          ! 判断每磁道扇区数是否=15
125     je      root_defined     ! 如果等于，则 ax 中就是引导驱动器的设备号。
126     mov     ax, #0x021c      ! /dev/PS0 - 1.44Mb

```

```

127      cmp     bx, #18
128      je      root_defined
129 undef_root:                                ! 如果都不一样, 则死循环(死机)。
130      jmp     undef_root
131 root_defined:
132      seg     cs
133      mov     root_dev, ax                  ! 将检查过的设备号保存起来。
134
135 ! after that (everything loaded), we jump to
136 ! the setup-routine loaded directly after
137 ! the bootblock:
! 到此, 所有程序都加载完毕, 我们就跳转到被
! 加载在 bootsect 后面的 setup 程序去。
138
139      jmp     0, SETUPSEG                  ! 跳转到 0x9020:0000 (setup.s 程序的开始处)。
                                           !!!!! 本程序到此就结束了。!!!!
! 下面是两个子程序。
140
141 ! This routine loads the system at address 0x10000, making sure
142 ! no 64kB boundaries are crossed. We try to load it as fast as
143 ! possible, loading whole tracks whenever we can.
144 !
145 ! in:  es - starting address segment (normally 0x1000)
146 !
! 该子程序将系统模块加载到内存地址 0x10000 处, 并确定没有跨越 64KB 的内存边界。我们试图尽快
! 地进行加载, 只要可能, 就每次加载整条磁道的数据。
! 输入:  es - 开始内存地址段值(通常是 0x1000)
147 sread:  .word 1+SETUPLEN                ! sectors read of current track
                                           ! 当前磁道中已读的扇区数。开始时已经读进 1 扇区的引导扇区
                                           ! bootsect 和 setup 程序所占的扇区数 SETUPLEN。
148 head:   .word 0                        ! current head   !当前磁头号。
149 track:   .word 0                        ! current track  !当前磁道号。
150
151 read_it:
! 测试输入的段值。从盘上读入的数据必须存放在位于内存地址 64KB 的边界开始处, 否则进入死循环。
! 清 bx 寄存器, 用于表示当前段内存放数据的开始位置。
152      mov     ax, es
153      test    ax, #0xffff
154 die:     jne     die                    ! es must be at 64kB boundary ! es 值必须位于 64KB 地址边界!
155      xor     bx, bx                    ! bx is starting address within segment ! bx 为段内偏移位置。
156 rp_read:
! 判断是否已经读入全部数据。比较当前所读段是否就是系统数据末端所处的段(#ENDSEG), 如果不是就
! 跳转至下面 ok1_read 标号处继续读数据。否则退出子程序返回。
157      mov     ax, es
158      cmp     ax, #ENDSEG                ! have we loaded all yet? ! 是否已经加载了全部数据?
159      jb      ok1_read
160      ret
161 ok1_read:
! 计算和验证当前磁道需要读取的扇区数, 放在 ax 寄存器中。
! 根据当前磁道还未读取的扇区数以及段内数据字节开始偏移位置, 计算如果全部读取这些未读扇区,
所
! 读总字节数是否会超过 64KB 段长度的限制。若会超过, 则根据此次最多能读入的字节数(64KB - 段
内

```


! 偏移位置), 反算出此次需要读取的扇区数。

```

162      seg cs
163      mov ax, sectors      ! 取每磁道扇区数。
164      sub ax, sread        ! 减去当前磁道已读扇区数。
165      mov cx, ax           ! cx = ax = 当前磁道未读扇区数。
166      shl cx, #9          ! cx = cx * 512 字节。
167      add cx, bx           ! cx = cx + 段内当前偏移值(bx)
                                ! = 此次读操作后, 段内共读入的字节数。
168      jnc ok2_read        ! 若没有超过 64KB 字节, 则跳转至 ok2_read 处执行。
169      je ok2_read
170      xor ax, ax           ! 若加上此次将读磁道上所有未读扇区时会超过 64KB, 则计算
171      sub ax, bx           ! 此时最多能读入的字节数(64KB - 段内读偏移位置), 再转换
172      shr ax, #9          ! 成需要读取的扇区数。
173 ok2_read:
174      call read_track
175      mov cx, ax           ! cx = 该次操作已读取的扇区数。
176      add ax, sread        ! 当前磁道上已经读取的扇区数。
177      seg cs
178      cmp ax, sectors      ! 如果当前磁道上的还有扇区未读, 则跳转到 ok3_read 处。
179      jne ok3_read
! 读该磁道的下一磁头面(1 号磁头)上的数据。如果已经完成, 则去读下一磁道。
180      mov ax, #1
181      sub ax, head         ! 判断当前磁头号。
182      jne ok4_read        ! 如果是 0 磁头, 则再去读 1 磁头面上的扇区数据。
183      inc track           ! 否则去读下一磁道。
184 ok4_read:
185      mov head, ax         ! 保存当前磁头号。
186      xor ax, ax           ! 清当前磁道已读扇区数。
187 ok3_read:
188      mov sread, ax        ! 保存当前磁道已读扇区数。
189      shl cx, #9          ! 上次已读扇区数*512 字节。
190      add bx, cx           ! 调整当前段内数据开始位置。
191      jnc rp_read         ! 若小于 64KB 边界值, 则跳转到 rp_read(156 行)处, 继续读数据。
                                ! 否则调整当前段, 为读下一段数据作准备。
192      mov ax, es
193      add ax, #0x1000      ! 将段基址调整为指向下一个 64KB 内存开始处。
194      mov es, ax
195      xor bx, bx           ! 清段内数据开始偏移值。
196      jmp rp_read         ! 跳转至 rp_read(156 行)处, 继续读数据。
197
! 读当前磁道上指定开始扇区和需读扇区数的数据到 es:bx 开始处。参见第 67 行下对 BIOS 磁盘读中断
! int 0x13, ah=2 的说明。
! al - 需读扇区数; es:bx - 缓冲区开始位置。
198 read_track:
199      push ax
200      push bx
201      push cx
202      push dx
203      mov dx, track        ! 取当前磁道号。
204      mov cx, sread        ! 取当前磁道上已读扇区数。
205      inc cx              ! cl = 开始读扇区。
206      mov ch, dl          ! ch = 当前磁道号。
207      mov dx, head        ! 取当前磁头号。

```



```

208      mov dh,dl          ! dh = 磁头号。
209      mov dl,#0          ! dl = 驱动器号(为 0 表示当前 A 驱动器)。
210      and dx,#0x0100     ! 磁头号不大于 1。
211      mov ah,#2          ! ah = 2, 读磁盘扇区功能号。
212      int 0x13
213      jc bad_rt          ! 若出错, 则跳转至 bad_rt。
214      pop dx
215      pop cx
216      pop bx
217      pop ax
218      ret
    ! 执行驱动器复位操作(磁盘中断功能号 0), 再跳转到 read_track 处重试。
219 bad_rt: mov ax,#0
220      mov dx,#0
221      int 0x13
222      pop dx
223      pop cx
224      pop bx
225      pop ax
226      jmp read_track
227
228 /*
229  * This procedure turns off the floppy drive motor, so
230  * that we enter the kernel in a known state, and
231  * don't have to worry about it later.
232  */
    ! 这个子程序用于关闭软驱的马达, 这样我们进入内核后它处于已知状态, 以后也就无须担心它了。
233 kill_motor:
234      push dx
235      mov dx,#0x3f2       ! 软驱控制卡的驱动端口, 只写。
236      mov al,#0          ! A 驱动器, 关闭 FDC, 禁止 DMA 和中断请求, 关闭马达。
237      outb               ! 将 al 中的内容输出到 dx 指定的端口去。
238      pop dx
239      ret
240
241 sectors:
242      .word 0             ! 存放当前启动软盘每磁道的扇区数。
243
244 msg1:
245      .byte 13,10         ! 回车、换行的 ASCII 码。
246      .ascii "Loading system ..."
247      .byte 13,10,13,10   ! 共 24 个 ASCII 码字符。
248
249 .org 508                ! 表示下面语句从地址 508(0x1FC)开始, 所以 root_dev
    ! 在启动扇区的第 508 开始的 2 个字节中。
250 root_dev:
251      .word ROOT_DEV      ! 这里存放根文件系统所在的设备号(init/main.c 中会
    ! 用)。
252 boot_flag:
253      .word 0xAA55        ! 硬盘有效标识。
254
255 .text
256 endtext:

```

```

257 .data
258 enddata:
259 .bss
260 endbss:

```

3.3.3 其它信息

对 bootsect.s 这段程序的说明和描述，在互连网上可以搜索到大量的资料。其中 Alessandro Rubini 著而由本人翻译的《Linux 内核源代码漫游》一篇文章(<http://oldlinux.org/Linux.old/docs/>)比较详细地描述了内核启动的详细过程，很有参考价值。由于这段程序是在 386 实模式下运行的，因此相对来说将比较容易理解。若此时阅读仍有困难，那么建议你首先再复习一下 80x86 汇编及其硬件的相关知识（可参阅参考文献[1]和[16]），然后再继续阅读本书。

对于最新开发的 Linux 内核，这段程序的改动也很小，基本保持了与 0.11 版的模样。

3.4 setup.s 程序

3.4.1 功能描述

setup 程序的作用主要是利用 ROM BIOS 中断读取机器系统数据，并将这些数据保存到 0x90000 开始的位置（覆盖掉了 bootsect 程序所在的地方），所取得的参数和保留的内存位置见下表 3-1 所示。这些参数将被内核中相关程序使用，例如字符设备驱动程序集中的 ttyio.c 程序等。

表 3-1 setup 程序读取并保留的参数

内存地址	长度(字节)	名称	描述
0x90000	2	光标位置	列号 (0x00-最左端)，行号 (0x00-最顶端)
0x90002	2	扩展内存数	系统从 1M 开始的扩展内存数值 (KB)。
0x90004	2	显示页面	当前显示页面
0x90006	1	显示模式	
0x90007	1	字符列数	
0x90008	2	??	
0x9000A	1	显示内存	显示内存(0x00-64k,0x01-128k,0x02-192k,0x03=256k)
0x9000B	1	显示状态	0x00-彩色,I/O=0x3dX; 0x11-单色,I/O=0x3bX
0x9000C	2	特性参数	显示卡特性参数
...			
0x90080	16	硬盘参数表	第 1 个硬盘的参数表
0x90090	16	硬盘参数表	第 2 个硬盘的参数表 (如果没有, 则清零)
0x901FC	2	根设备号	根文件系统所在的设备号 (bootsec.s 中设置)

然后 setup 程序将 system 模块从 0x10000-0x8ffff（当时认为内核系统模块 system 的长度不会超过此值：512KB）整块向下移动到内存绝对地址 0x00000 处。接下来加载中断描述符表寄存器(idtr)和全局描述符表寄存器(gdtr)，开启 A20 地址线，重新设置两个中断控制芯片 8259A，将硬件中断号重新设置为 0x20 - 0x2f。最后设置 CPU 的控制寄存器 CR0（也称机器状态字），从而进入 32 位保护模式运行，并跳转到位于 system 模块最前面部分的 head.s 程序继续运行。

为了能让 head.s 在 32 位保护模式下运行，在本程序中临时设置了中断描述符表 (idt) 和全局描述

符表 (gdt)，并在 gdt 中设置了当前内核代码段的描述符和数据段的描述符。在下面的 head.s 程序中会根据内核的需要重新设置这些描述符表。

现在，我们根据 CPU 在实模式和保护模式下寻址方式的不同，用比较的方法来简单说明 32 位保护模式运行机制的主要特点，以便能顺利地理解本节的程序。在后续章节中将逐步对其进行详细说明。

在实模式下，寻址一个内存地址主要是使用段和偏移值，段值被存放在段寄存器中（例如 ds），并且段的最大长度被固定为 64KB。段内偏移地址存放在任意一个可用于寻址的寄存器中（例如 si）。因此，根据段寄存器和偏移寄存器中的值，就可以算出实际指向的内存地址，见图 3-2 (a) 所示。

而在保护模式运行方式下，段寄存器中存放的不再是寻址段的基地址，而是一个段描述符表中某项的索引值。索引值指定的段描述符项中含有需要寻址的内存段的基地址、段的最大长度值和段的访问级别等信息。寻址的内存位置是由该段描述符项中指定的段基地址值和偏移值组合而成，段的最大长度也由描述符指定。可见，和实模式下的寻址相比，段寄存器值换成了段描述符索引，但偏移值还是原实模式下的概念。这样，在保护模式下寻址一个内存地址就需要比实模式下多一道手续，也即需要使用段描述符表。这是由于在保护模式下访问一个内存段需要的信息比较多，一个 16 位的段寄存器放不下这么多内容。示意图见图 3-2 (b) 所示。

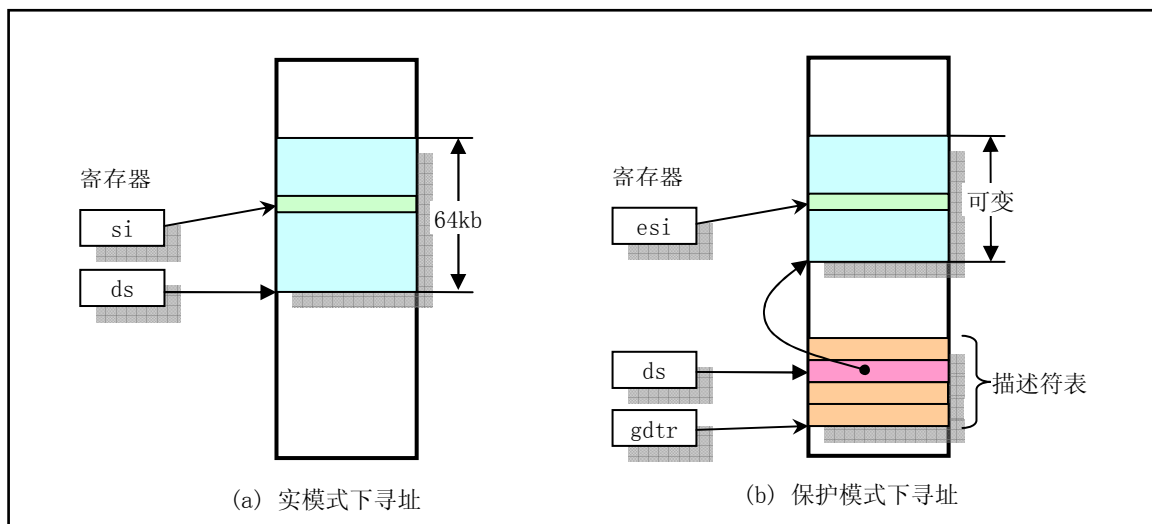


图 3-2 实模式与保护模式下寻址方式的比较

因此，在进入保护模式之前，必须首先设置好将要用到的段描述符表，例如全局描述符表 gdt。然后使用指令 lgdt 把描述符表的基地址告知 CPU (gdt 表的基地址存入 gdttr 寄存器)。再将机器状态字的保护模式标志置位即可进入 32 位保护运行模式。

3.4.2 代码注释

程序 3-2 linux/boot/setup.s

```

1 !
2 !      setup.s      (C) 1991 Linus Torvalds
3 !
4 ! setup.s is responsible for getting the system data from the BIOS,
5 ! and putting them into the appropriate places in system memory.
6 ! both setup.s and system has been loaded by the bootblock.
7 !

```

```

8 ! This code asks the bios for memory/disk/other parameters, and
9 ! puts them in a "safe" place: 0x90000-0x901FF, ie where the
10 ! boot-block used to be. It is then up to the protected mode
11 ! system to read them from there before the area is overwritten
12 ! for buffer-blocks.
!
! setup.s 负责从 BIOS 中获取系统数据，并将这些数据放到系统内存的适当地方。
! 此时 setup.s 和 system 已经由 bootsect 引导块加载到内存中。
!
! 这段代码询问 bios 有关内存/磁盘/其它参数，并将这些参数放到一个
! “安全的”地方：0x90000-0x901FF，也即原来 bootsect 代码块曾经在
! 的地方，然后在被缓冲块覆盖掉之前由保护模式的 system 读取。
13 !
14
15 ! NOTE! These had better be the same as in bootsect.s!
! 以下这些参数最好和 bootsect.s 中的相同！
16
17 INITSEG = 0x9000      ! we move boot here - out of the way ! 原来 bootsect 所处的段。
18 SYSSEG  = 0x1000      ! system loaded at 0x10000 (65536). ! system 在 0x10000 (64k) 处。
19 SETUPSEG = 0x9020     ! this is the current segment ! 本程序所在的段地址。
20
21 .globl begtext, begdata, begbss, endtext, enddata, endbss
22 .text
23 begtext:
24 .data
25 begdata:
26 .bss
27 begbss:
28 .text
29
30 entry start
31 start:
32
33 ! ok, the read went well so we get current cursor position and save it for
34 ! posterity.
! ok, 整个读磁盘过程都正常，现在将光标位置保存以备今后使用。
35
36      mov     ax, #INITSEG      ! this is done in bootsect already, but...
                                   ! 将 ds 置成 #INITSEG (0x9000)。这已经在 bootsect 程序中
                                   ! 设置过，但是现在是 setup 程序，Linus 觉得需要再重新
                                   ! 设置一下。
37      mov     ds, ax
38      mov     ah, #0x03        ! read cursor pos
                                   ! BIOS 中断 0x10 的读光标功能号 ah = 0x03
                                   ! 输入：bh = 页号
                                   ! 返回：ch = 扫描开始线，cl = 扫描结束线，
                                   ! dh = 行号 (0x00 是顶端)，dl = 列号 (0x00 是左边)。
39      xor     bh, bh
40      int     0x10             ! save it in known place, con_init fetches
41      mov     [0], dx          ! it from 0x90000.
                                   ! 上两句是说将光标位置信息存放在 0x90000 处，控制台
                                   ! 初始化时会来取。
42

```

```

43 ! Get memory size (extended mem, kB) ! 下面 3 句取扩展内存的大小值 (KB)。
      ! 是调用中断 0x15, 功能号 ah = 0x88
      ! 返回: ax = 从 0x100000 (1M) 处开始的扩展内存大小 (KB)。
      ! 若出错则 CF 置位, ax = 出错码。

44
45     mov     ah, #0x88
46     int     0x15
47     mov     [2], ax      ! 将扩展内存数值存在 0x90002 处 (1 个字)。
48
49 ! Get video-card data:      ! 下面这段用于取显卡当前显示模式。
      ! 调用 BIOS 中断 0x10, 功能号 ah = 0x0f
      ! 返回: ah = 字符列数, al = 显示模式, bh = 当前显示页。
      ! 0x90004 (1 字) 存放当前页, 0x90006 显示模式, 0x90007 字符列数。

50
51     mov     ah, #0x0f
52     int     0x10
53     mov     [4], bx      ! bh = display page
54     mov     [6], ax      ! al = video mode, ah = window width
55
56 ! check for EGA/VGA and some config parameters ! 检查显示方式 (EGA/VGA) 并取参数。
      ! 调用 BIOS 中断 0x10, 附加功能选择 - 取方式信息
      ! 功能号: ah = 0x12, bl = 0x10
      ! 返回: bh = 显示状态
      !         (0x00 - 彩色模式, I/O 端口=0x3dX)
      !         (0x01 - 单色模式, I/O 端口=0x3bX)
      ! bl = 安装的显示内存
      ! (0x00 - 64k, 0x01 - 128k, 0x02 - 192k, 0x03 = 256k)
      ! cx = 显卡特性参数 (参见程序后的说明)。

57
58     mov     ah, #0x12
59     mov     bl, #0x10
60     int     0x10
61     mov     [8], ax      ! 0x90008 = ??
62     mov     [10], bx     ! 0x9000A = 安装的显示内存, 0x9000B = 显示状态 (彩色/单色)
63     mov     [12], cx     ! 0x9000C = 显卡特性参数。
64
65 ! Get hd0 data      ! 取第一个硬盘的信息 (复制硬盘参数表)。
      ! 第 1 个硬盘参数表的首地址竟然是中断向量 0x41 的向量值! 而第 2 个硬盘
      ! 参数表紧接第 1 个表的后面, 中断向量 0x46 的向量值也指向这第 2 个硬盘
      ! 的参数表首址。表的长度是 16 个字节 (0x10)。
      ! 下面两段程序分别复制 BIOS 有关两个硬盘的参数表, 0x90080 处存放第 1 个
      ! 硬盘的表, 0x90090 处存放第 2 个硬盘的表。

66
67     mov     ax, #0x0000
68     mov     ds, ax
69     lds     si, [4*0x41] ! 取中断向量 0x41 的值, 也即 hd0 参数表的地址 → ds:si
70     mov     ax, #INITSEG
71     mov     es, ax
72     mov     di, #0x0080 ! 传输的目的地址: 0x9000:0x0080 → es:di
73     mov     cx, #0x10   ! 共传输 0x10 字节。
74     rep
75     movsb
76

```

```

77 ! Get hdl data
78
79     mov     ax, #0x0000
80     mov     ds, ax
81     lds     si, [4*0x46] ! 取中断向量 0x46 的值, 也即 hdl 参数表的地址 → ds:si
82     mov     ax, #INITSEG
83     mov     es, ax
84     mov     di, #0x0090 ! 传输的目的地址: 0x9000:0x0090 → es:di
85     mov     cx, #0x10
86     rep
87     movsb
88
89 ! Check that there IS a hdl :- ) ! 检查系统是否存在第 2 个硬盘, 如果不存在则第 2 个表清零。
! 利用 BIOS 中断调用 0x13 的取盘类型功能。
! 功能号 ah = 0x15;
! 输入: dl = 驱动器号 (0x8X 是硬盘: 0x80 指第 1 个硬盘, 0x81 第 2 个硬盘)
! 输出: ah = 类型码; 00 -- 没有这个盘, CF 置位; 01 -- 是软驱, 没有 change-line 支持;
!       02 -- 是软驱 (或其它可移动设备), 有 change-line 支持; 03 -- 是硬盘。
90
91     mov     ax, #0x01500
92     mov     dl, #0x81
93     int     0x13
94     jc      no_disk1
95     cmp     ah, #3 ! 是硬盘吗? (类型 = 3 ? )。
96     je      is_disk1
97 no_disk1:
98     mov     ax, #INITSEG ! 第 2 个硬盘不存在, 则对第 2 个硬盘表清零。
99     mov     es, ax
100    mov     di, #0x0090
101    mov     cx, #0x10
102    mov     ax, #0x00
103    rep
104    stosb
105 is_disk1:
106
107 ! now we want to move to protected mode ... ! 现在我们要进入保护模式中了...
108
109     cli ! no interrupts allowed ! ! 此时不允许中断。
110
111 ! first we move the system to it's rightful place
! 首先我们将 system 模块移到正确的位置。
! bootsect 引导程序是将 system 模块读入到从 0x10000 (64k) 开始的位置。由于当时假设
! system 模块最大长度不会超过 0x80000 (512k), 也即其末端不会超过内存地址 0x90000,
! 所以 bootsect 会将自己移动到 0x90000 开始的地方, 并把 setup 加载到它的后面。
! 下面这段程序的用途是再把整个 system 模块移动到 0x00000 位置, 即把从 0x10000 到 0x8ffff
! 的内存数据块 (512k), 整块地向内存低端移动了 0x10000 (64k) 的位置。
112
113     mov     ax, #0x0000
114     cld ! 'direction'=0, movs moves forward
115 do_move:
116     mov     es, ax ! destination segment ! es:di → 目的地址 (初始为 0x0000:0x0)
117     add     ax, #0x1000
118     cmp     ax, #0x9000 ! 已经把从 0x8000 段开始的 64k 代码移动完?

```

```

119      jz      end_move
120      mov     ds,ax          ! source segment ! ds:si→源地址(初始为 0x1000:0x0)
121      sub     di,di
122      sub     si,si
123      mov     cx,#0x8000    ! 移动 0x8000 字 (64k 字节)。
124      rep
125      movsw
126      jmp     do_move
127
128 ! then we load the segment descriptors
! 此后，我们加载段描述符。
! 从这里开始会遇到 32 位保护模式的操作，因此需要 Intel 32 位保护模式编程方面的知识了，
! 有关这方面的信息请查阅列表后的简单介绍或附录中的详细说明。这里仅作概要说明。
! 在进入保护模式中运行之前，我们需要首先设置好需要使用的段描述符表。这里需要设置全局
! 描述符表和中断描述符表。
!
! lidt 指令用于加载中断描述符表(idt)寄存器，它的操作数是 6 个字节，0-1 字节是描述符表的
! 长度值(字节)；2-5 字节是描述符表的 32 位线性基地址(首地址)，其形式参见下面
! 219-220 行和 223-224 行的说明。中断描述符表中的每一个表项(8 字节)指出发生中断时
! 需要调用的代码的信息，与中断向量有些相似，但要包含更多的信息。
!
! lgdt 指令用于加载全局描述符表(gdt)寄存器，其操作数格式与 lidt 指令的相同。全局描述符
! 表中的每个描述符项(8 字节)描述了保护模式下数据和代码段(块)的信息。其中包括段的
! 最大长度限制(16 位)、段的线性基址(32 位)、段的特权级、段是否在内存、读写许可以及
! 其它一些保护模式运行的标志。参见后面 205-216 行。
!
129
130 end_move:
131      mov     ax,#SETUPSEG   ! right, forgot this at first. didn't work :-)
132      mov     ds,ax          ! ds 指向本程序(setup)段。
133      lidt    idt_48         ! load idt with 0,0
! 加载中断描述符表(idt)寄存器，idt_48 是 6 字节操作数的位置
! (见 218 行)。前 2 字节表示 idt 表的限长，后 4 字节表示 idt 表
! 所处的基地址。
134      lgdt    gdt_48        ! load gdt with whatever appropriate
! 加载全局描述符表(gdt)寄存器，gdt_48 是 6 字节操作数的位置
! (见 222 行)。
135
136 ! that was painless, now we enable A20
! 以上的操作很简单，现在我们开启 A20 地址线。参见程序列表后有关 A20 信号线的说明。
! 关于所涉及的一些端口和命令，可参考 kernel/chr_drv/keyboard.S 程序后对键盘接口的说明。
137
138      call    empty_8042     ! 等待输入缓冲器空。
! 只有当输入缓冲器为空时才可以对其进行写命令。
139      mov     al,#0xD1       ! command write ! 0xD1 命令码-表示要写数据到
140      out     #0x64,al        ! 8042 的 P2 端口。P2 端口的位 1 用于 A20 线的选通。
! 数据要写到 0x60 口。
141      call    empty_8042     ! 等待输入缓冲器空，看命令是否被接受。
142      mov     al,#0xDF       ! A20 on ! 选通 A20 地址线的参数。
143      out     #0x60,al
144      call    empty_8042     ! 输入缓冲器为空，则表示 A20 线已经选通。
145
146 ! well, that went ok, I hope. Now we have to reprogram the interrupts :-(

```



```

147 ! we put them right after the intel-reserved hardware interrupts, at
148 ! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
149 ! messed this up with the original PC, and they haven't been able to
150 ! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
151 ! which is used for the internal hardware interrupts as well. We just
152 ! have to reprogram the 8259's, and it isn't fun.
!! 希望以上一切正常。现在我们必须重新对中断进行编程⊗
!! 我们将它们放在正好处于 intel 保留的硬件中断后面, 在 int 0x20-0x2F。
!! 在那里它们不会引起冲突。不幸的是 IBM 在原 PC 机中搞糟了, 以后也没有纠正过来。
!! PC 机的 bios 将中断放在了 0x08-0x0f, 这些中断也被用于内部硬件中断。
!! 所以我们就必须重新对 8259 中断控制器进行编程, 这一点都没劲。

153
154      mov     al, #0x11                ! initialization sequence
                                         ! 0x11 表示初始化命令开始, 是 ICW1 命令字, 表示边
                                         ! 沿触发、多片 8259 级连、最后要发送 ICW4 命令字。
155      out     #0x20, al                ! send it to 8259A-1 ! 发送到 8259A 主芯片。
! 下面定义的两个字是直接使用机器码表示的两条相对跳转指令, 起延时作用。
! 0xeb 是直接近跳转指令的操作码, 带 1 个字节的相对位移值。因此跳转范围是-127 到 127。CPU 通过
! 把这个相对位移值加到 EIP 寄存器中就形成一个新的有效地址。此时 EIP 指向下一条被执行的指令。
! 执行时所花费的 CPU 时钟周期数是 7 至 10 个。0x00eb 表示跳转值是 0 的一条指令, 因此还是直接
! 执行下一条指令。这两条指令共可提供 14-20 个 CPU 时钟周期的延迟时间。在 as86 中没有表示相应
! 指令的助记符, 因此 Linus 在 setup.s 等一些汇编程序中就直接使用机器码来表示这种指令。另外,
! 每个空操作指令 NOP 的时钟周期数是 3 个, 因此若要达到相同的延迟效果就需要 6 至 7 个 NOP 指令。
156      .word   0x00eb, 0x00eb          ! jmp $+2, jmp $+2 ! '$' 表示当前指令的地址,
157      out     #0xA0, al                ! and to 8259A-2 ! 再发送到 8259A 从芯片。
158      .word   0x00eb, 0x00eb
159      mov     al, #0x20                ! start of hardware int's (0x20)
160      out     #0x21, al                ! 送主芯片 ICW2 命令字, 起始中断号, 要送奇地址。
161      .word   0x00eb, 0x00eb
162      mov     al, #0x28                ! start of hardware int's 2 (0x28)
163      out     #0xA1, al                ! 送从芯片 ICW2 命令字, 从芯片的起始中断号。
164      .word   0x00eb, 0x00eb
165      mov     al, #0x04                ! 8259-1 is master
166      out     #0x21, al                ! 送主芯片 ICW3 命令字, 主芯片的 IR2 连从芯片 INT。
167      .word   0x00eb, 0x00eb          ! 参见代码列表后的说明。
168      mov     al, #0x02                ! 8259-2 is slave
169      out     #0xA1, al                ! 送从芯片 ICW3 命令字, 表示从芯片的 INT 连到主芯
                                         ! 片的 IR2 引脚上。
170      .word   0x00eb, 0x00eb
171      mov     al, #0x01                ! 8086 mode for both
172      out     #0x21, al                ! 送主芯片 ICW4 命令字。8086 模式; 普通 EOI 方式,
                                         ! 需发送指令来复位。初始化结束, 芯片就绪。
173      .word   0x00eb, 0x00eb
174      out     #0xA1, al                ! 送从芯片 ICW4 命令字, 内容同上。
175      .word   0x00eb, 0x00eb
176      mov     al, #0xFF                ! mask off all interrupts for now
177      out     #0x21, al                ! 屏蔽主芯片所有中断请求。
178      .word   0x00eb, 0x00eb
179      out     #0xA1, al                ! 屏蔽从芯片所有中断请求。
180
181 ! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
182 ! need no steenking BIOS anyway (except for the initial loading :-).
183 ! The BIOS-routine wants lots of unnecessary data, and it's less

```

```

184 ! "interesting" anyway. This is how REAL programmers do it.
185 !
186 ! Well, now's the time to actually move into protected mode. To make
187 ! things as simple as possible, we do no register set-up or anything,
188 ! we let the gnu-compiled 32-bit programs do that. We just jump to
189 ! absolute address 0x00000, in 32-bit protected mode.
!! 哼，上面这段当然没劲☹，希望这样能工作，而且我们也不再需要乏味的 BIOS 了（除了
!! 初始的加载☹。BIOS 子程序要求很多不必要的数 据，而且它一点都没趣。那是“真正”的
!! 程序员所做的事。
190
! 这里设置进入 32 位保护模式运行。首先加载机器状态字(lmsw-Load Machine Status Word)，也称
! 控制寄存器 CR0，其比特位 0 置 1 将导致 CPU 工作在保护模式。
191     mov     ax, #0x0001      ! protected mode (PE) bit ! 保护模式比特位(PE)。
192     lmsw    ax              ! This is it!! 就这样加载机器状态字!
193     jmp     0, 8            ! jmp offset 0 of segment 8 (cs) ! 跳转至 cs 段 8，偏移 0 处。
! 我们已经将 system 模块移动到 0x00000 开始的地方，所以这里的偏移地址是 0。这里的段
! 值的 8 已经是保护模式下的段选择符了，用于选择描述符表和描述符表项以及所要求的特权级。
! 段选择符长度为 16 位（2 字节）；位 0-1 表示请求的特权级 0-3，linux 操作系统只
! 用到两级：0 级（系统级）和 3 级（用户级）；位 2 用于选择全局描述符表(0)还是局部描
! 述符表(1)；位 3-15 是描述符表项的索引，指出选择第几项描述符。所以段选择符
! 8(0b0000, 0000, 0000, 1000)表示请求特权级 0、使用全局描述符表中的第 1 项，该项指出
! 代码的基地址是 0（参见 209 行），因此这里的跳转指令就会去执行 system 中的代码。
194
195 ! This routine checks that the keyboard command queue is empty
196 ! No timeout is used - if this hangs there is something wrong with
197 ! the machine, and we probably couldn't proceed anyway.
! 下面这个子程序检查键盘命令队列是否为空。这里不使用超时方法 - 如果这里死机，
! 则说明 PC 机有问题，我们就没有办法再处理下去了。
! 只有当输入缓冲器为空时（状态寄存器位 2 = 0）才可以对其进行写命令。
198 empty_8042:
199     .word   0x00eb, 0x00eb   ! 这是两个跳转指令的机器码(跳转到下一句)，相当于延时空操作。
200     in      al, #0x64        ! 8042 status port ! 读 AT 键盘控制器状态寄存器。
201     test    al, #2           ! is input buffer full? ! 测试位 2，输入缓冲器满？
202     jnz     empty_8042      ! yes - loop
203     ret
204
205 gdt: ! 全局描述符表开始处。描述符表由多个 8 字节长的描述符项组成。
! 这里给出了 3 个描述符项。第 1 项无用（206 行），但须存在。第 2 项是系统代码段
! 描述符（208-211 行），第 3 项是系统数据段描述符(213-216 行)。每个描述符的具体
! 含义参见列表后说明。
206     .word   0, 0, 0, 0      ! dummy ! 第 1 个描述符，不用。
207 ! 这里在 gdt 表中的偏移量为 0x08，当加载代码段寄存器(段选择符)时，使用的是这个偏移值。
208     .word   0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
209     .word   0x0000          ! base address=0
210     .word   0x9A00          ! code read/exec
211     .word   0x00C0          ! granularity=4096, 386
212 ! 这里在 gdt 表中的偏移量是 0x10，当加载数据段寄存器(如 ds 等)时，使用的是这个偏移值。
213     .word   0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
214     .word   0x0000          ! base address=0
215     .word   0x9200          ! data read/write
216     .word   0x00C0          ! granularity=4096, 386
217
218 idt_48:

```

```

219      .word    0                ! idt limit=0
220      .word    0,0             ! idt base=0L
221
222 gdt_48:
223      .word    0x800            ! gdt limit=2048, 256 GDT entries
                                ! 全局表长度为 2k 字节，因为每 8 字节组成一个段描述符项
                                ! 所以表中共可有 256 项。
224      .word    512+gdt,0x9      ! gdt base = 0X9xxxx
                                ! 4 个字节构成的内存线性地址：0x0009<<16 + 0x0200+gdt
                                ! 也即 0x90200 + gdt(即在本程序段中的偏移地址，205 行)。
225
226 .text
227 endtext:
228 .data
229 enddata:
230 .bss
231 endbss:

```

3.4.3 其它信息

为了获取机器的基本参数，这段程序多次调用了 BIOS 中的中断，并开始涉及一些对硬件端口的操作。下面简要地描述了程序中使用到的 BIOS 中断调用，并对 A20 地址线问题的原由进行了解释，最后提及关于 Intel 32 位保护模式运行的问题。

3.4.3.1 当前内存映像

在 setup.s 程序执行结束后，系统模块 system 被移动到物理地址 0x0000 开始处，而从 0x90000 处则存放了内核将会使用的一些系统基本参数，示意图如图 3-3 所示。

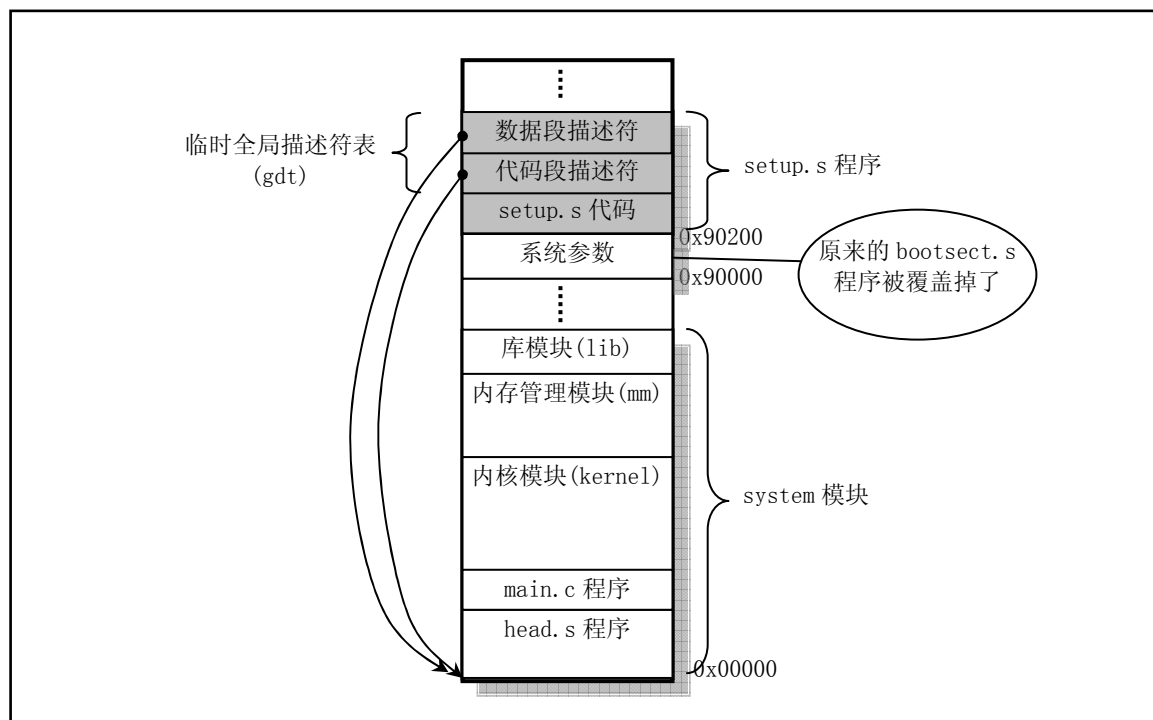


图 3-3 setup.s 程序结束后内存中程序示意图

此时临时全局表中有三个描述符，第一个是（NULL）不用，另外两个分别是代码段描述符和数据段描述符。它们都指向系统模块的起始处，也即物理地址 0x0000 处。这样当 setup.s 中执行最后一条指令 `jmp 0,8`（第 193 行）时，就会跳到 head.s 程序开始处继续执行下去。这条指令中的 '8' 是段选择符，用来指定所需使用的描述符项，此处是指 gdt 中的代码段描述符。'0' 是描述符项指定的代码段中的偏移值。

3.4.3.2 BIOS 视频中断 0x10

这里说明上面程序中用到的 ROM BIOS 中视频中断调用的几个子功能。

A. 获取显示卡信息（其它辅助功能选择）：

表 3-2 获取显示卡信息（功能号： ah = 0x12, bh = 0x10）

输入/返回信息	寄存器	内容说明
输入信息	ah	功能号=0x12，获取显示卡信息
	bh	子功能号=0x10。
返回信息	bh	视频状态： 0x00 – 彩色模式（此时视频硬件 I/O 端口基地址为 0x3DX）； 0x01 – 单色模式（此时视频硬件 I/O 端口基地址为 0x3BX）； 注：其中端口地址中的 X 值可为 0–f。
	bl	已安装的显示内存大小： 00 = 64K, 01 = 128K, 02 = 192K, 03 = 256K
	ch	特性连接器比特位信息： 比特位 说明 0 特性线 1，状态 2； 1 特性线 0，状态 2； 2 特性线 1，状态 1； 3 特性线 0，状态 1； 4-7 未使用(为 0)
	cl	视频开关设置信息： 比特位 说明 0 开关 1 关闭； 1 开关 2 关闭； 2 开关 3 关闭； 3 开关 4 关闭； 4-7 未使用。 原始 EGA/VGA 开关设置值： 0x00 MDA/HGC； 0x01-0x03 MDA/HGC； 0x04 CGA 40x25； 0x05 CGA 80x25； 0x06 EGA+ 40x25； 0x07-0x09 EGA+ 80x25； 0x0A EGA+ 80x25 单色； 0x0B EGA+ 80x25 单色。

3.4.3.3 硬盘基本参数表 (“INT 0x41”)

中断向量表中, int 0x41 的中断向量位置 ($4 * 0x41 = 0x0000:0x0104$) 存放的并不是中断程序的地址, 而是第一个硬盘的基本参数表。对于 100% 兼容的 BIOS 来说, 这里存放着硬盘参数表阵列的首地址 F000h:E401h。第二个硬盘的基本参数表入口地址存于 int 0x46 中断向量位置处。

表 3-3 硬盘基本参数信息表

位移	大小	英文名称	说明
0x00	字	cyl	柱面数
0x02	字节	head	磁头数
0x03	字		开始减小写电流的柱面(仅 PC XT 使用, 其它为 0)
0x05	字	wpcom	开始写前预补偿柱面号 (乘 4)
0x07	字节		最大 ECC 猝发长度 (仅 XT 使用, 其它为 0)
0x08	字节	ctl	控制字节 (驱动器步进选择) 位 0 未用 位 1 保留(0) (关闭 IRQ) 位 2 允许复位 位 3 若磁头数大于 8 则置 1 位 4 未用(0) 位 5 若在柱面数+1 处有生产商的坏区图, 则置 1 位 6 禁止 ECC 重试 位 7 禁止访问重试。
0x09	字节		标准超时值 (仅 XT 使用, 其它为 0)
0x0A	字节		格式化超时值 (仅 XT 使用, 其它为 0)
0x0B	字节		检测驱动器超时值 (仅 XT 使用, 其它为 0)
0x0C	字	lzone	磁头着陆(停止)柱面号
0x0E	字节	sect	每磁道扇区数
0x0F	字节		保留。

3.4.3.4 A20 地址线问题

1981 年 8 月, IBM 公司最初推出的个人计算机 IBM PC 使用的 CPU 是 Intel 8088。在该微机中地址线只有 20 根(A0 – A19)。在当时内存 RAM 只有几百 KB 或不到 1MB 时, 20 根地址线已足够用来寻址这些内存。其所能寻址的最高地址是 0xffff:0xffff, 也即 0x10ffef。对于超出 0x100000(1MB)的寻址地址将默认地环绕到 0x0ffef。当 IBM 公司于 1985 年引入 AT 机时, 使用的是 Intel 80286 CPU, 具有 24 根地址线, 最高可寻址 16MB, 并且有一个与 8088 完全兼容的实模式运行方式。然而, 在寻址值超过 1MB 时它却不能象 8088 那样实现地址寻址的环绕。但是当时已经有一些程序是利用这种地址环绕机制进行工作的。为了实现完全的兼容性, IBM 公司发明了使用一个开关来开启或禁止 0x100000 地址比特位。由于在当时的 8042 键盘控制器上恰好有空闲的端口引脚 (输出端口 P2, 引脚 P21), 于是便使用了该引脚来作为与门控制这个地址比特位。该信号即被称为 A20。如果它为零, 则比特 20 及以上地址都被清除。从而实现了兼容性。

由于在机器启动时, 默认条件下, A20 地址线是禁止的, 所以操作系统必须使用适当的方法来开启它。但是由于各种兼容机所使用的芯片集不同, 要做到这一点却是非常的麻烦。因此通常要在几种控制方法中选择。

对 A20 信号线进行控制的常用方法是通过设置键盘控制器的端口值。这里的 setup.s 程序 (138-144 行)即使用了这种典型的控制方式。对于其它一些兼容微机还可以使用其它方式来做对 A20 线的控制。

有些操作系统将 A20 的开启和禁止作为实模式与保护运行模式之间进行转换的标准过程中的一部分。由于键盘的控制器速度很慢，因此就不能使用键盘控制器对 A20 线来进行操作。为此引进了一个 A20 快速门选项(Fast Gate A20)，它使用 I/O 端口 0x92 来处理 A20 信号线，避免了使用慢速的键盘控制器操作方式。对于不含键盘控制器的系统就只能使用 0x92 端口来控制，但是该端口也有可能被其它兼容微机上的设备（如显示芯片）所使用，从而造成系统错误的操作。

还有一种方式是通过读 0xee 端口来开启 A20 信号线，写该端口则会禁止 A20 信号线。

3.4.3.5 8259 中断控制器芯片

8259A 是一种可编程的中断控制芯片，每片可以管理 8 个中断源。通过多片的级联方式，能构成最多管理 64 个中断向量的系统。在 PC/AT 系列兼容机中，使用了两片 8259A 芯片，共可管理 15 级中断向量。其级连示意图见图 3-4 所示。其中从芯片的 INT 引脚连接到主芯片的 IR2 引脚上。主 8259A 芯片的端口基地址是 0x20，从芯片是 0xA0。

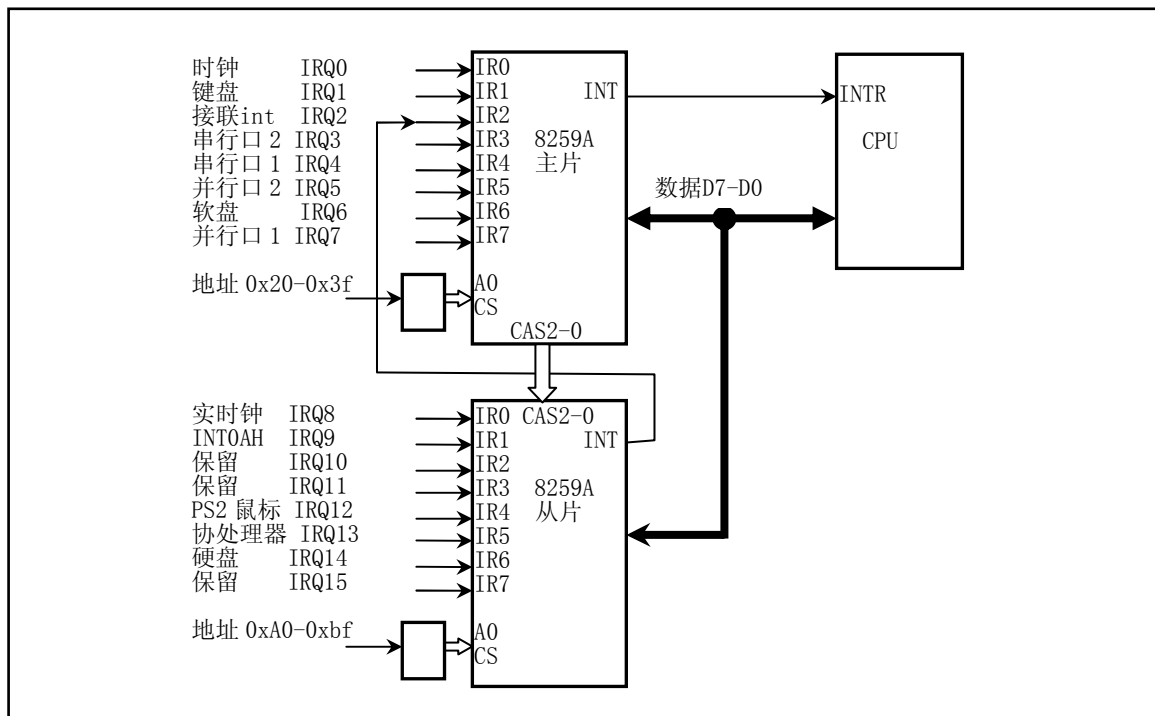


图 3-4 PC/AT 微机级连式 8259 控制系统

在总线控制器的控制下，芯片可以处于编程状态和操作状态。编程状态是 CPU 使用 IN 或 OUT 指令对 8259A 芯片进行初始化编程的状态。一旦完成了初始化编程，芯片即进入操作状态，此时芯片即可随时响应外部设备提出的中断请求（IRQ0 - IRQ15）。通过中断判优选择，芯片将选中当前最高优先级的中断请求作为中断服务对象，并通过 CPU 引脚 INT 通知 CPU 外中断请求的到来，CPU 响应后，芯片从数据总线 D7-D0 将编程设定的当前服务对象的中断号送出，CPU 由此获取对应的中断向量值，并执行中断服务程序。

在 Linux 内核中，这些硬件中断信号对应的中断号是从 int 32（0x20）开始的（int 0 - int 31 被用于 CPU 的陷阱中断），也即中断号范围是 int32 -- int 47。

3.4.3.6 Intel CPU 32 位保护运行模式

Intel CPU 一般可以在两种模式下运行，即实地址模式和保护模式。早期的 Intel CPU（8088/8086）只能工作在实模式下，某一时刻只能运行单个任务。对于 Intel 80386 以上的芯片则还可以运行在 32 位保护模式下。在保护模式下运行可以支持多任务；支持 4G 的物理内存；支持虚拟内存；支持内存的页

式管理和段式管理；支持特权级。

虽然对保护模式下的运行机制是理解 Linux 内核的重要基础，但由于篇幅所限，对其工作原理的简单介绍可以参考书后的附录。但仍然建议初学者能够对书后列出的相关书籍，作一番仔细研究。为了真正理解 setup.s 程序和下面 head.s 程序的作用，起码要先明白段选择符、段描述符和 80x86 的页表寻址机制。

3.4.3.7 内存管理寄存器

Intel 80386 CPU 有 4 个寄存器用来定位控制分段内存管理的数据结构：

GDTR （Global Descriptor Table Register）全局描述符表寄存器；

LDTR （Local Descriptor Table Register）局部描述符表寄存器；

这两个寄存器用于指向段描述符表 GDT 和 LDT，对这两个表的详细说明请参见附录。

IDTR （Interrupt Descriptor Table Register）中断描述符表寄存器；

这个寄存器指向中断处理向量（句柄）表（IDT）的入口点。所有中断处理过程的入口地址信息均存放在该表中的描述符表项中。

TR （Task Register）任务寄存器；

该寄存器指向处理器定义当前任务（进程）所需的信息，也即任务数据结构 task{ }。

3.4.3.8 控制寄存器

Intel 80386 的控制寄存器共有 4 个，分别命名为 CR0、CR1、CR2、CR3。这些寄存器仅能够由系统程序通过 MOV 指令访问。见图 3-5 所示。

31		23		15		7		0						
页目录基址寄存器 Page Directory Base Register (PDBR)						保留 Reserved				CR3				
页异常线性地址 Page Fault Linear Address														
保留 Reserved										CR2				
保留 Reserved														
保留 Reserved										CR1				
保留 Reserved														
P	保留 Reserved								E	T	E	M	P	CR0
G	保留 Reserved								T	S	M	P	E	

图 3-5 控制寄存器结构

控制寄存器 CR0 含有系统整体的控制标志，它控制或指示出整个系统的运行状态或条件。其中：

- ♦ PE – 保护模式开启位（Protection Enable，比特位 0）。如果设置了该比特位，就会使处理器开始在保护模式下运行。
- ♦ MP – 协处理器存在标志（Math Present，比特位 1）。用于控制 WAIT 指令的功能，以配合协处理的运行。
- ♦ EM – 仿真控制（Emulation，比特位 2）。指示是否需要仿真协处理器的功能。
- ♦ TS – 任务切换（Task Switch，比特位 3）。每当任务切换时处理器就会设置该比特位，并且在解释协处理器指令之前测试该位。
- ♦ ET – 扩展类型（Extention Type，比特位 4）。该位指出了系统中所含有的协处理器类型（是 80287 还是 80387）。
- ♦ PG – 分页操作（Paging，比特位 31）。该位指示出是否使用页表将线性地址变换成物理地址。参见第 10 章对分页内存管理的描述。

CR2 用于 PG 置位时处理页异常操作。CPU 会将引起错误的线性地址保存在该寄存器中。

CR3 同样也是在 PG 标志置位时起作用。该寄存器为 CPU 指定当前运行的任务所使用的页表目录。

3.5 head.s 程序

3.5.1 功能描述

head.s 程序在被编译后, 会被连接成 system 模块的最前面开始部分, 这也就是为什么称其为头部(head)程序的原因。从这里开始, 内核完全都是在保护模式下运行了。heads.s 汇编程序与前面的语法格式不同, 它采用的是 AT&T 的汇编语言格式, 并且需要使用 GNU 的 gas 和 gld2 进行编译连接。因此请注意代码中赋值的方向是从左到右。

这段程序实际上处于内存绝对地址 0 处开始的地方。这个程序的功能比较单一。首先是加载各个数据段寄存器, 重新设置中断描述符表 idt, 共 256 项, 并使各个表项均指向一个只报错误的哑中断程序。然后重新设置全局描述符表 gdt。接着使用物理地址 0 与 1M 开始处的内容相比较的方法, 检测 A20 地址线是否已真的开启(如果没有开启, 则在访问高于 1Mb 物理内存地址时 CPU 实际只会访问 (IP MOD 1Mb) 地址处的内容), 如果检测下来发现没有开启, 则进入死循环。然后程序测试 PC 机是否含有数学协处理器芯片(80287、80387 或其兼容芯片), 并在控制寄存器 CR0 中设置相应的标志位。接着设置管理内存的分页处理机制, 将页目录表放在绝对物理地址 0 开始处(也是本程序所处的物理内存位置, 因此这段程序将被覆盖掉), 紧随后面放置共可寻址 16MB 内存的 4 个页表, 并分别设置它们的表项。最后利用返回指令将预先放置在堆栈中的/init/main.c 程序的入口地址弹出, 去运行 main() 程序。

3.5.2 代码注释

程序 3-3 linux/boot/head.s

```

1  /*
2  *  linux/boot/head.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  head.s contains the 32-bit startup code.
9  *
10 *  NOTE!!! Startup happens at absolute address 0x00000000, which is also where
11 *  the page directory will exist. The startup code will be overwritten by
12 *  the page directory.
13 */
14 /*
15 *  head.s 含有 32 位启动代码。
16 *  注意!!! 32 位启动代码是从绝对地址 0x00000000 开始的, 这里也同样是页目录将存在的地方,
17 *  因此这里的启动代码将被页目录覆盖掉。
18 */
19 .text
20 .globl _idt, _gdt, _pg_dir, _tmp_floppy_area
21 _pg_dir:  # 页目录将会存放在这里。
22 startup_32:  # 18-22 行设置各个数据段寄存器。
23     movl $0x10, %eax  # 对于 GNU 汇编来说, 每个直接数要以 '$' 开始, 否则是表示地址。

```

2 在当前的 Linux 操作系统中, gas 和 gld 已经分别更名为 as 和 ld。

每个寄存器名都要以'%'开头, eax 表示是 32 位的 ax 寄存器。

再次注意!!! 这里已经处于 32 位运行模式, 因此这里的\$0x10 并不是把地址 0x10 装入各个
段寄存器, 它现在其实是全局段描述符表中的偏移值, 或者更正确地说是一个描述符表项
的选择符。有关选择符的说明请参见 setup.s 中 193 行下的说明。这里\$0x10 的含义是请求
特权级 0 (位 0-1=0)、选择全局描述符表 (位 2=0)、选择表中第 2 项 (位 3-15=2)。它正好指
向表中的数据段描述符项。(描述符的具体数值参见前面 setup.s 中 212, 213 行)
下面代码的含义是: 置 ds, es, fs, gs 中的选择符为 setup.s 中构造的数据段 (全局段描述符表
的第 2 项) =0x10, 并将堆栈放置在 stack_start 指向的 user_stack 数组区, 然后使用本程序
后面定义的新中断描述符表和全局段描述表。新全局段描述表中初始内容与 setup.s 中的基本
一样, 仅段限长从 8MB 修改成了 16MB。stack_start 定义在 kernel/sched.c, 69 行。它是指向
user_stack 数组末端的一个长指针。

```

19      mov %ax, %ds
20      mov %ax, %es
21      mov %ax, %fs
22      mov %ax, %gs
23      lss _stack_start, %esp    # 表示_stack_start→ss:esp, 设置系统堆栈。
                                # stack_start 定义在 kernel/sched.c, 69 行。
24      call setup_idt           # 调用设置中断描述符表子程序。
25      call setup_gdt           # 调用设置全局描述符表子程序。
26      movl $0x10, %eax         # reload all the segment registers
27      mov %ax, %ds             # after changing gdt. CS was already
28      mov %ax, %es             # reloaded in 'setup_gdt'
29      mov %ax, %fs             # 因为修改了 gdt, 所以需要重新装载所有的段寄存器。
30      mov %ax, %gs             # CS 代码段寄存器已经在 setup_gdt 中重新加载过了。
# 由于段描述符中的段限长从 setup.s 程序的 8MB 改成了本程序设置的 16MB (见 setup.s 行 208-216
# 和本程序后面的行 235-236), 因此这里再次对所有段寄存器执行加载操作是必须的。
# 另外, 通过使用 bochs 跟踪观察, 如果不对 CS 再次执行加载, 那么在执行到行 26 时 CS 代码段不可见
# 部分中的限长还是 8MB。这样看来应该重新加载 CS, 但在实际机器上测试结果表明 CS 已经加载过了。
31      lss _stack_start, %esp
# 32-36 行用于测试 A20 地址线是否已经开启。采用的方法是向内存地址 0x000000 处写入任意
# 一个数值, 然后看内存地址 0x100000 (1M) 处是否也是这个数值。如果一直相同的话, 就一直
# 比较下去, 也即死循环、死机。表示地址 A20 线没有选通, 结果内核就不能使用 1M 以上内存。
32      xorl %eax, %eax
33 1:    incl %eax                # check that A20 really IS enabled
34      movl %eax, 0x000000      # loop forever if it isn't
35      cmpl %eax, 0x100000
36      je 1b                    # '1b' 表示向后 (backward) 跳转到标号 1 去 (33 行)。
                                # 若是 '5f' 则表示向前 (forward) 跳转到标号 5 去。
37 /*
38  * NOTE! 486 should set bit 16, to check for write-protect in supervisor
39  * mode. Then it would be unnecessary with the "verify_area()" -calls.
40  * 486 users probably want to set the NE (#5) bit also, so as to use
41  * int 16 for math errors.
42  */
/*
 * 注意! 在下面这段程序中, 486 应该将位 16 置位, 以检查在超级用户模式下的写保护,
 * 此后 "verify_area()" 调用中就不需要了。486 的用户通常也会想将 NE (#5) 置位, 以便
 * 对数学协处理器的出错使用 int 16。
 */
# 下面这段程序 (43-65) 用于检查数学协处理器芯片是否存在。方法是修改控制寄存器 CR0, 在
# 假设存在协处理器的情况下执行一个协处理器指令, 如果出错的话则说明协处理器芯片不存在,
# 需要设置 CR0 中的协处理器仿真位 EM (位 2), 并复位协处理器存在标志 MP (位 1)。
```

```

43      movl %cr0,%eax      # check math chip
44      andl $0x80000011,%eax # Save PG,PE,ET
45 /* "orl $0x10020,%eax" here for 486 might be good */
46      orl $2,%eax        # set MP
47      movl %eax,%cr0
48      call check_x87
49      jmp after_page_tables # 跳转到 135 行。
50
51 /*
52 * We depend on ET to be correct. This checks for 287/387.
53 */
54 /*
55 * 我们依赖于 ET 标志的正确性来检测 287/387 存在与否。
56 */
57 check_x87:
58      fninit
59      fstsw %ax
60      cmpb $0,%al
61      je 1f              /* no coprocessor: have to set bits */
62      movl %cr0,%eax      # 如果存在的则向前跳转到标号 1 处，否则改写 cr0。
63      xorl $6,%eax        /* reset MP, set EM */
64      movl %eax,%cr0
65      ret
66
67 .align 2 # 这里".align 2"的含义是指存储边界对齐调整。"2"表示调整到地址最后 2 位为零，
68          # 即按 4 字节方式对齐内存地址。
69 1:      .byte 0xDB,0xE4    /* fsetpm for 287, ignored by 387 */ # 287 协处理器码。
70      ret
71
72 /*
73 * setup_idt
74 *
75 * sets up a idt with 256 entries pointing to
76 * ignore_int, interrupt gates. It then loads
77 * idt. Everything that wants to install itself
78 * in the idt-table may do so themselves. Interrupts
79 * are enabled elsewhere, when we can be relatively
80 * sure everything is ok. This routine will be over-
81 * written by the page tables.
82 */
83 /*
84 * 下面这段是设置中断描述符表子程序 setup_idt
85 *
86 * 将中断描述符表 idt 设置成具有 256 个项，并都指向 ignore_int 中断门。然后加载中断
87 * 描述符表寄存器(用 lidt 指令)。真正实用的中断门以后再安装。当我们在其它地方认为一切
88 * 都正常时再开启中断。该子程序将会被页表覆盖掉。
89 */
90 # 中断描述符表中的项虽然也是 8 字节组成，但其格式与全局表中的不同，被称为门描述符
91 # (Gate Descriptor)。它的 0-1,6-7 字节是偏移量，2-3 字节是选择符，4-5 字节是一些标志。
92
93 setup_idt:
94      lea ignore_int,%edx  # 将 ignore_int 的有效地址（偏移值）值→edx 寄存器
95      movl $0x00080000,%eax # 将选择符 0x0008 置入 eax 的高 16 位中。
96      movw %dx,%ax         /* selector = 0x0008 = cs */
97                          # 偏移值的低 16 位置入 eax 的低 16 位中。此时 eax 含有

```

```

82          movw $0x8E00,%dx          #门描述符低 4 字节的值。
83                                     /* interrupt gate - dpl=0, present */
84          lea _idt,%edi              # 此时 edx 含有门描述符高 4 字节的值。
85          mov $256,%ecx              # _idt 是中断描述符表的地址。
86 rp_sidt:
87          movl %eax, (%edi)           # 将哑中断门描述符存入表中。
88          movl %edx, 4(%edi)
89          addl $8,%edi                # edi 指向表中下一项。
90          dec %ecx
91          jne rp_sidt
92          lidt idt_descr              # 加载中断描述符表寄存器值。
93          ret
94
95 /*
96 *  setup_gdt
97 *
98 *  This routines sets up a new gdt and loads it.
99 *  Only two entries are currently built, the same
100 *  ones that were built in init.s. The routine
101 *  is VERY complicated at two whole lines, so this
102 *  rather long comment is certainly needed :-).
103 *  This routine will beoverwritten by the page tables.
104 */
105 /*
106 * 设置全局描述符表项 setup_gdt
107 * 这个子程序设置一个新的全局描述符表 gdt，并加载。此时仅创建了两个表项，与前
108 * 面的一样。该子程序只有两行，“非常的”复杂，所以当然需要这么长的注释了☺。
109 * 该子程序将被页表覆盖掉。
110 */
111 setup_gdt:
112          lgdt gdt_descr              # 加载全局描述符表寄存器(内容已设置好，见 234-238 行)。
113          ret
114
115 /*
116 *  I put the kernel page tables right after the page directory,
117 *  using 4 of them to span 16 Mb of physical memory. People with
118 *  more than 16MB will have to expand this.
119 */
120 /* Linus 将内核的内存页表直接放在页目录之后，使用了 4 个表来寻址 16 Mb 的物理内存。
121 * 如果你有多于 16 Mb 的内存，就需要在这里进行扩充修改。
122 */
123 # 每个页表长为 4 Kb 字节（1 页内存页面），而每个页表项需要 4 个字节，因此一个页表共可以存放
124 # 1024 个表项。如果一个页表项寻址 4 Kb 的地址空间，则一个页表就可以寻址 4 Mb 的物理内存。
125 # 页表项的格式为：项的前 0-11 位存放一些标志，例如是否在内存中(P 位 0)、读写许可(R/W 位 1)、
126 # 普通用户还是超级用户使用(U/S 位 2)、是否修改过(是否脏了)(D 位 6)等；表项的位 12-31 是
127 # 页框地址，用于指出一页内存的物理起始地址。
128
129 .org 0x1000          # 从偏移 0x1000 处开始是第 1 个页表（偏移 0 开始处将存放页表目录）。
130 pg0:
131
132 .org 0x2000
133 pg1:
134

```

```

120 .org 0x3000
121 pg2:
122
123 .org 0x4000
124 pg3:
125
126 .org 0x5000      # 定义下面的内存数据块从偏移 0x5000 处开始。
127 /*
128  * tmp_floppy_area is used by the floppy-driver when DMA cannot
129  * reach to a buffer-block. It needs to be aligned, so that it isn't
130  * on a 64kB border.
131  */
132 /* 当 DMA（直接存储器访问）不能访问缓冲块时，下面的 tmp_floppy_area 内存块
133  * 就可供软盘驱动程序使用。其地址需要对齐调整，这样就不会跨越 64kB 边界。
134  */
135 _tmp_floppy_area:
136     .fill 1024, 1, 0      # 共保留 1024 项，每项 1 字节，填充数值 0。
137
138 # 下面这几个入栈操作 (pushl) 用于为调用 /init/main.c 程序和返回作准备。
139 # 前面 3 个入栈 0 值应该分别是 envp、argv 指针和 argc 值，但 main() 没有用到。
140 # 139 行的入栈操作是模拟调用 main.c 程序时首先将返回地址入栈的操作，所以如果
141 # main.c 程序真的退出时，就会返回到这里的标号 L6 处继续执行下去，也即死循环。
142 # 140 行将 main.c 的地址压入堆栈，这样，在设置分页处理 (setup_paging) 结束后
143 # 执行 'ret' 返回指令时就会将 main.c 程序的地址弹出堆栈，并去执行 main.c 程序去了。
144 after_page_tables:
145     pushl $0              # These are the parameters to main :- )
146     pushl $0              # 这些是调用 main 程序的参数（指 init/main.c）。
147     pushl $0              # 其中的 '$' 符号表示这是一个立即操作数。
148     pushl $L6             # return address for main, if it decides to.
149     pushl $_main          # '_main' 是编译程序对 main 的内部表示方法。
150     jmp setup_paging      # 跳转至第 198 行。
151 L6:
152     jmp L6                # main should never return here, but
153                             # just in case, we know what happens.
154
155 /* This is the default interrupt "handler" :- ) */
156 /* 下面是默认的中断“向量句柄” ☺ */
157 int_msg:
158     .asciz "Unknown interrupt\n\r"      # 定义字符串“未知中断(回车换行)”。
159     .align 2                            # 按 4 字节方式对齐内存地址。
160 ignore_int:
161     pushl %eax
162     pushl %ecx
163     pushl %edx
164     push %ds      # 这里请注意！！ds, es, fs, gs 等虽然是 16 位的寄存器，但入栈后
165                     # 仍然会以 32 位的形式入栈，也即需要占用 4 个字节的堆栈空间。
166     push %es
167     push %fs
168     movl $0x10, %eax  # 置段选择符（使 ds, es, fs 指向 gdt 表中的数据段）。
169     mov %ax, %ds
170     mov %ax, %es
171     mov %ax, %fs
172     pushl $int_msg    # 把调用 printk 函数的参数指针（地址）入栈。

```

```

162      call _printk      # 该函数在/kernel/printk.c 中。
                        # ' _printk' 是 printk 编译后模块中的内部表示法。
163      popl %eax
164      pop %fs
165      pop %es
166      pop %ds
167      popl %edx
168      popl %ecx
169      popl %eax
170      iret             # 中断返回（把中断调用时压入栈的 CPU 标志寄存器（32 位）值也弹出）。
171
172
173 /*
174  * Setup_paging
175  *
176  * This routine sets up paging by setting the page bit
177  * in cr0. The page tables are set up, identity-mapping
178  * the first 16MB. The pager assumes that no illegal
179  * addresses are produced (ie >4Mb on a 4Mb machine).
180  *
181  * NOTE! Although all physical memory should be identity
182  * mapped by this routine, only the kernel page functions
183  * use the >1Mb addresses directly. All "normal" functions
184  * use just the lower 1Mb, or the local data space, which
185  * will be mapped to some other place - mm keeps track of
186  * that.
187  *
188  * For those with more memory than 16 Mb - tough luck. I've
189  * not got it, why should you :-). The source is here. Change
190  * it. (Seriously - it shouldn't be too difficult. Mostly
191  * change some constants etc. I left it at 16Mb, as my machine
192  * even cannot be extended past that (ok, but it was cheap :-).
193  * I've tried to show which constants to change by having
194  * some kind of marker at them (search for "16Mb"), but I
195  * won't guarantee that's all :-( )
196  */
/*
 * 这个子程序通过设置控制寄存器 cr0 的标志（PG 位 31）来启动对内存的分页处理功能，
 * 并设置各个页表项的内容，以恒等映射前 16 MB 的物理内存。分页器假定不会产生非法的
 * 地址映射（也即在只有 4Mb 的机器上设置出大于 4Mb 的内存地址）。
 * 注意！尽管所有的物理地址都应该由这个子程序进行恒等映射，但只有内核页面管理函数能
 * 直接使用>1Mb 的地址。所有“一般”函数仅使用低于 1Mb 的地址空间，或者是使用局部数据
 * 空间，地址空间将被映射到其它一些地方去 -- mm(内存管理程序)会管理这些事的。
 * 对于那些有多于 16Mb 内存的家伙 - 真是太幸运了，我还没有，为什么你会有☺。代码就在
 * 这里，对它进行修改吧。（实际上，这并不太困难的。通常只需修改一些常数等。我把它设置
 * 为 16Mb，因为我的机器再怎么扩充甚至不能超过这个界限（当然，我的机器是很便宜的☺）。
 * 我已经通过设置某类标志来给出需要改动的地方（搜索“16Mb”），但我不能保证作这些
 * 改动就行了☺）。
 */
# 在内存物理地址 0x0 处开始存放 1 页页目录表和 4 页页表。页目录表是系统所有进程公用的，而
# 这里的 4 页页表则是属于内核专用。对于新的进程，系统会在主内存区为其申请页面存放页表。
# 1 页内存长度是 4096 字节。
197 .align 2             # 按 4 字节方式对齐内存地址边界。

```



```

198 setup_paging:  # 首先对 5 页内存 (1 页目录 + 4 页页表) 清零
199     movl $1024*5,%ecx          /* 5 pages - pg_dir+4 page tables */
200     xorl %eax,%eax
201     xorl %edi,%edi             /* pg_dir is at 0x000 */
                                # 页目录从 0x000 地址开始。
202     cld;rep;stosl
# 下面 4 句设置页目录表中的项, 因为我们 (内核) 共有 4 个页表所以只需设置 4 项。
# 页目录项的结构与页表中项的结构一样, 4 个字节为 1 项。参见上面 113 行下的说明。
# "$pg0+7" 表示: 0x00001007, 是页目录表中的第 1 项。
# 则第 1 个页表所在的地址 = 0x00001007 & 0xfffff000 = 0x1000;
# 第 1 个页表的属性标志 = 0x00001007 & 0x00000fff = 0x07, 表示该页存在、用户可读写。
203     movl $pg0+7,_pg_dir        /* set present bit/user r/w */
204     movl $pg1+7,_pg_dir+4      /* ----- " " ----- */
205     movl $pg2+7,_pg_dir+8      /* ----- " " ----- */
206     movl $pg3+7,_pg_dir+12     /* ----- " " ----- */
# 下面 6 行填写 4 个页表中所有项的内容, 共有: 4(页表)*1024(项/页表)=4096 项(0 - 0xfff),
# 也即能映射物理内存 4096*4Kb = 16Mb。
# 每项的内容是: 当前项所映射的物理内存地址 + 该页的标志 (这里均为 7)。
# 使用的方法是从最后一个页表的最后一项开始按倒退顺序填写。一个页表的最后一项在页表中的
# 位置是 1023*4 = 4092。因此最后一页的最后一项的位置就是$pg3+4092。
207     movl $pg3+4092,%edi        # edi → 最后一页的最后一项。
208     movl $0xffff007,%eax       /* 16Mb - 4096 + 7 (r/w user,p) */
                                # 最后 1 项对应物理内存页面的地址是 0xffff000,
                                # 加上属性标志 7, 即为 0xffff007。
209     std                        # 方向位置位, edi 值递减(4 字节)。
210 1:     stosl                   /* fill pages backwards - more efficient :- ) */
211     subl $0x1000,%eax          # 每填写好一项, 物理地址值减 0x1000。
212     jge 1b                    # 如果小于 0 则说明全添写好了。
# 设置页目录基址寄存器 cr3 的值, 指向页目录表。
213     xorl %eax,%eax            /* pg_dir is at 0x0000 */    # 页目录表在 0x0000 处。
214     movl %eax,%cr3            /* cr3 - page directory start */
# 设置启动使用分页处理 (cr0 的 PG 标志, 位 31)
215     movl %cr0,%eax
216     orl $0x80000000,%eax      # 添上 PG 标志。
217     movl %eax,%cr0            /* set paging (PG) bit */
218     ret                       /* this also flushes prefetch-queue */
# 在改变分页处理标志后要求使用转移指令刷新预取指令队列, 这里用的是返回指令 ret。
# 该返回指令的另一个作用是将堆栈中的 main 程序的地址弹出, 并开始运行/init/main.c 程序。
# 本程序到此真正结束了。
219
220 .align 2                      # 按 4 字节方式对齐内存地址边界。
221 .word 0
222 idt_descr:                    # 下面两行是 lidt 指令的 6 字节操作数: 长度, 基址。
223     .word 256*8-1             # idt contains 256 entries
224     .long _idt
225 .align 2
226 .word 0
227 gdt_descr:                    # 下面两行是 lgdt 指令的 6 字节操作数: 长度, 基址。
228     .word 256*8-1             # so does gdt (not that that's any      # not → note.
229     .long _gdt                # magic number, but it works for me :)
230
231 .align 3                      # 按 8 字节方式对齐内存地址边界。
232 _idt:     .fill 256,8,0       # idt is uninitialized    # 256 项, 每项 8 字节, 填 0。

```



```
233 # 全局表。前 4 项分别是空项（不用）、代码段描述符、数据段描述符、系统段描述符，其中
234 # 系统段描述符 linux 没有派用处。后面还预留了 252 项的空间，用于放置所创建任务的
235 # 局部描述符(LDT)和对应的任务状态段 TSS 的描述符。
236 # (0-nul, 1-cs, 2-ds, 3-sys, 4-TSS0, 5-LDT0, 6-TSS1, 7-LDT1, 8-TSS2 etc...)
237 _gdt: .quad 0x0000000000000000 /* NULL descriptor */
238      .quad 0x00c09a0000000fff /* 16Mb */      # 0x08, 内核代码段最大长度 16M。
239      .quad 0x00c0920000000fff /* 16Mb */      # 0x10, 内核数据段最大长度 16M。
240      .quad 0x0000000000000000 /* TEMPORARY - don't use */
241      .fill 252, 8, 0 /* space for LDT's and TSS's etc */
```

3.5.3 其它信息

3.5.3.1 程序执行结束后的内存映像

head.s 程序执行结束后，已经正式完成了内存页目录和页表的设置，并重新设置了内核实际使用的中断描述符表 idt 和全局描述符表 gdt。另外还为软盘驱动程序开辟了 1KB 字节的缓冲区。此时 system 模块在内存中的详细映像见图 3-6 所示。

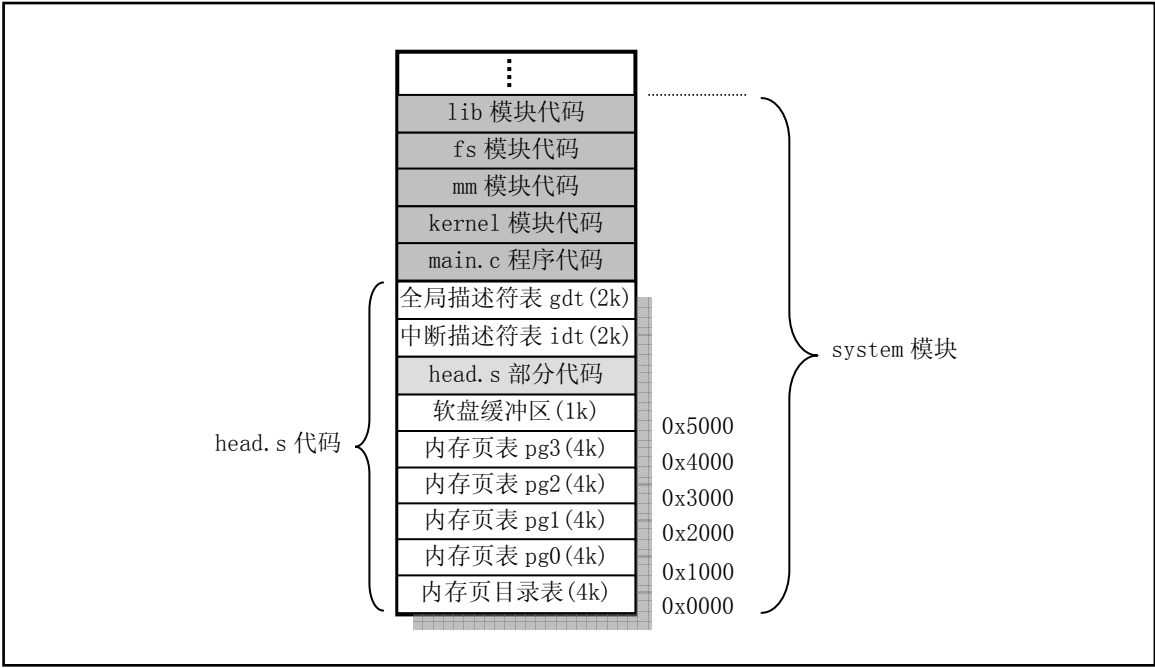


图 3-6 system 模块在内存中的映像示意图

3.5.3.2 Intel 32 位保护运行机制

理解这段程序的关键是真正了解 Intel 386 32 位保护模式的运行机制，也是继续阅读以下其余程序所必须的。为了与 8086 CPU 兼容，80x86 的保护模式被处理的较为复杂。当 CPU 运行在保护模式下时，它就将实模式下的段地址当作保护模式下段描述符的指针使用，此时段寄存器中存放的是一个描述符在描述符表中的偏移地址值。而当前描述符表的基地址则保存在描述符表寄存器中，如全局描述符表寄存器 gdt、中断门描述符表寄存器 idtr，加载这些表寄存器须使用专用指令 lgdt 或 lidt。

CPU 在实模式运行方式时，段寄存器用来放置一个内存段地址（比如 0x9000），而此时在该段内可以寻址 64KB 的内存。但当进入保护模式运行方式时，此时段寄存器中放置的并不是内存中的某个地址值，而是指定描述符表中某个描述符项相对于该描述符表基址的一个偏移量。在这个 8 字节的描述符中含有该段线性地址的‘段’基址和段的长度，以及其它一些描述该段特征的比特位。因此此时所寻址的内存位置是这个段基址加上当前执行代码指针 eip 的值。当然，此时所寻址的实际物理内存地址，还需

要经过内存页面处理管理机制进行变换后才能得到。简而言之，32 位保护模式下的内存寻址需要拐个弯，经过描述符表中的描述符和内存页管理来确定。

针对不同的使用方面，描述符表分为三种：全局描述符表（GDT）、中断描述符表（IDT）和局部描述符表（LDT）。当 CPU 运行在保护模式下，某一时刻 GDT 和 IDT 分别只能有一个，分别由寄存器 GDTR 和 IDTR 指定它们的表基址。局部表可以有 0-8191 个，其基址由当前 LDTR 寄存器的内容指定，是使用 GDT 中某个描述符来加载的，也即 LDT 也是由 GDT 中的描述符来指定。但是在某一时刻同样也只有其中的一个被认为是活动的。一般对于每个任务（进程）使用一个 LDT。在运行时，程序可以使用 GDT 中的描述符以及当前任务的 LDT 中的描述符。

中断描述符表 IDT 的结构与 GDT 类似，在 Linux 内核中它正好位于 GDT 表的后面。共含有 256 项 8 字节的描述符。但每个描述符项的格式与 GDT 的不同，其中存放着相应中断过程的偏移值（0-1，6-7 字节）、所处段的选择符值（2-3 字节）和一些标志（4-5 字节）。

图 3-7 是 Linux 内核中所使用的描述符表在内存中的示意图。图中，每个任务在 GDT 中占有两个描述符项。GDT 表中的 LDT0 描述符项是第一个任务（进程）的局部描述符表的描述符，TSS0 是第一个任务的任务状态段（TSS）的描述符。每个 LDT 中含有三个描述符，其中第一个不用，第二个是任务代码段的描述符，第三个是任务数据段和堆栈段的描述符。当 DS 段寄存器中是第一个任务的数据段选择符时，DS:ESI 即指向该任务数据段中的某个数据。

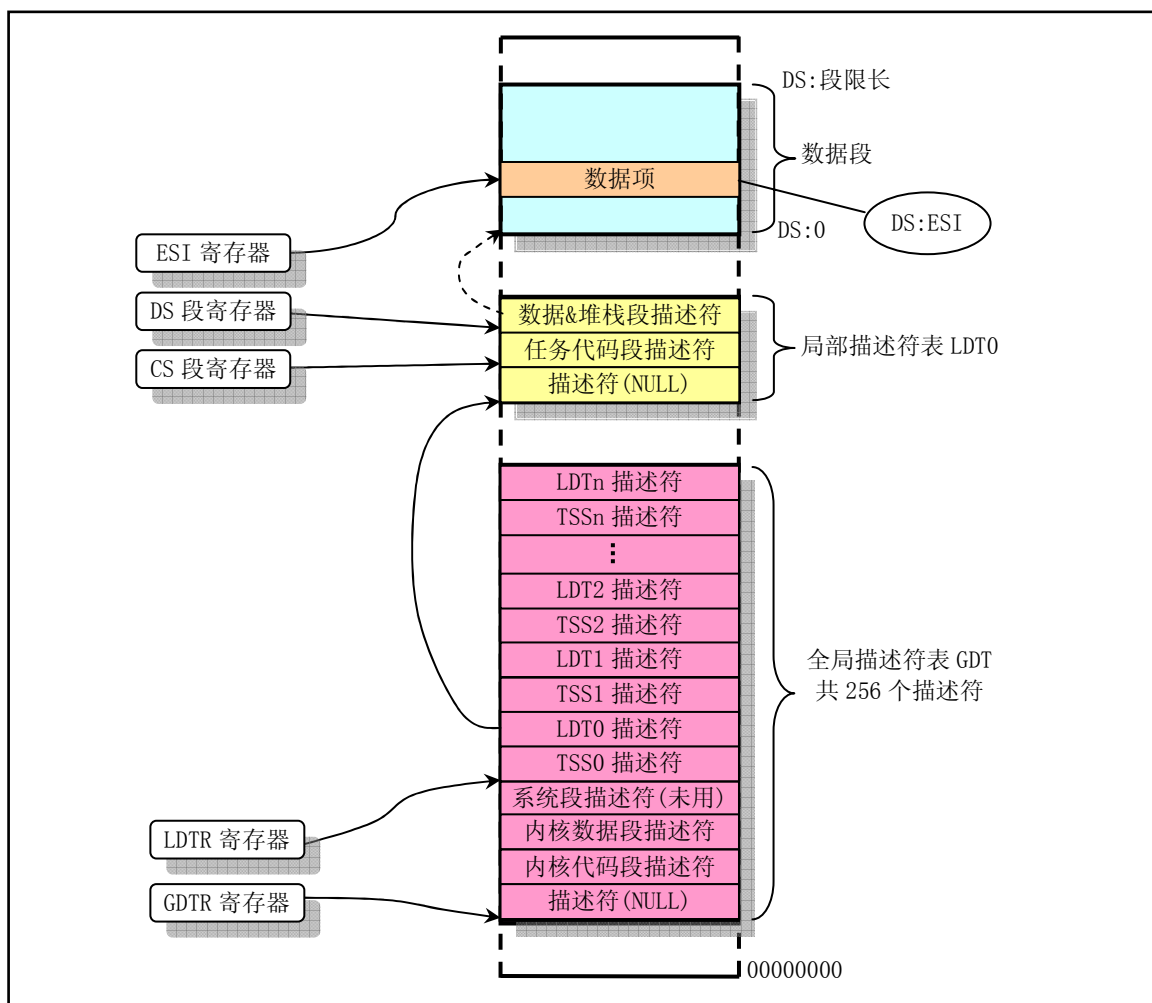


图 3-7 Linux 内核使用描述符表的示意图。

3.6 本章小结

在引导加载程序 bootsect.s 将 setup.s 代码和 system 模块加载到内存中，并且分别把自己和 setup.s 代码移动到物理内存 0x90000 和 0x90200 处后，就把执行权交给了 setup 程序。其中 system 模块的首部包含有 head.s 代码。

setup 程序的主要作用是利用 ROM BIOS 的中断程序获取机器的一些基本参数，并保存在 0x90000 开始的内存块中，供后面程序使用。同时把 system 模块往下移动到物理地址 0x00000 开始处，这样，system 中的 head.s 代码就处在 0x00000 开始处了。然后加载描述符表基地址到描述符表寄存器中，为进行 32 位保护模式下的运行作好准备。接下来对中断控制硬件进行重新设置，最后通过设置机器控制寄存器 CR0 并跳转到 system 模块的 head.s 代码开始处，使 CPU 进入 32 位保护模式下运行。

Head.s 代码的主要作用是初步初始化中断描述符表中的 256 项门描述符，检查 A20 地址线是否已经打开，测试系统是否含有数学协处理器。然后初始化内存页目录表，为内存的分页管理作好准备工作。最后跳转到 system 模块中的初始化程序 init.c 中继续执行。

下一章的主要内容就是详细描述 init/main.c 程序的功能和作用。