程序 6-3 linux/boot/head.s

```
/*
 *  linux/boot/head.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  head.s contains the 32-bit startup code.
 *
 * NOTE!!! Startup happens at absolute address 0x00000000, which is also where
 * the page directory will exist. The startup code will be overwritten by
 * the page directory.
 */
/*
 *  head.s 含有 32 位启动代码。
 * 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，
 * 因此这里的启动代码将被页目录覆盖掉。
 */
.text
.globl _idt,_gdt,_pg_dir,_tmp_floppy_area
_pg_dir:                    # 页目录将会存放在这里。

# 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，或者更准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：设置 ds,es,fs,gs 为 setup.s 中构造的数据段（全局段描述符表第 2 项）
# 的选择符=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。第 23 行设置这里使用的栈，姑且称为系统栈。但在移动到
# 任务 0 执行（init/main.c 中 137 行）以后该栈就被用作任务 0 和任务 1 共同使用的用户栈了。

startup_32:                 # 18-22 行设置各个数据段寄存器。
        movl $0x10,%eax     # 对于 GNU 汇编，每个直接操作数要以'$'开始，否则表示地址。
                            # 每个寄存器名都要以'%'开头，eax 表示是 32 位的 ax 寄存器。
        mov %ax,%ds
        mov %ax,%es
        mov %ax,%fs
        mov %ax,%gs
        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。
        call setup_idt              # 调用设置中断描述符表子程序。
        call setup_gdt              # 调用设置全局描述符表子程序。
        movl $0x10,%eax             # reload all the segment registers
        mov %ax,%ds                 # after changing gdt. CS was already
        mov %ax,%es                 # reloaded in 'setup_gdt'
        mov %ax,%fs                 # 因为修改了 gdt，所以需要重新装载所有的段寄存器。
        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 中的 8MB 改成了本程序设置的 16MB（见 setup.s 行 208-216
```

```
# 和本程序后面的 235-236 行），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 26 行时 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。但是由于 setup.s 中的内核代码段描述符与本程序中
# 重新设置的代码段描述符除了段限长以外其余部分完全一样，8MB 的限长在内核初始化阶段不会有
# 问题，而且在以后内核执行过程中段间跳转时会重新加载 CS。因此这里没有加载它并没有让程序
# 出错。
# 针对该问题，目前内核中就在第 25 行之后添加了一条长跳转指令：'ljmp $(__KERNEL_CS),$1f'，
# 跳转到第 26 行来确保 CS 确实又被重新加载。

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 1MB 以上内存。
#
# 33 行上的'1:'是一个局部符号构成的标号。标号由符号后跟一个冒号组成。此时该符号表示活动
# 位置计数（Active location counter）的当前值，并可以作为指令的操作数。局部符号用于帮助
# 编译器和编程人员临时使用一些名称。共有 10 个局部符号名，可在整个程序中重复使用。这些符号
# 名使用名称'0'、'1'、...、'9'来引用。为了定义一个局部符号，需把标号写成'N:'形式（其中 N
# 表示一个数字）。为了引用先前最近定义的这个符号，需要写成'Nb'，其中 N 是定义标号时使用的
# 数字。为了引用一个局部标号的下一个定义，需要写成'Nf'，这里 N 是 10 个前向引用之一。上面
# 'b'表示“向后（backwards）”，'f'表示“向前（forwards）”。在汇编程序的某一处，我们最大
# 可以向后/向前引用 10 个标号（最远第 10 个）。

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。
 */
# 上面原注释中提到的 486 CPU 中 CR0 控制寄存器的位 16 是写保护标志 WP（Write-Protect），
# 用于禁止超级用户级的程序向一般用户只读页面中进行写操作。该标志主要用于操作系统在创建
# 新进程时实现写时复制（copy-on-write）方法。
# 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   */
    /*
     * 我们依赖于 ET 标志的正确性来检测 287/387 存在与否。
     */
    # 下面 fninit 和 fstsw 是数学协处理器（80287/80387）的指令。
    # finit 向协处理器发出初始化命令，它会把协处理器置于一个未受以前操作影响的已知状态，设置
    # 其控制字为默认值、清除状态字和所有浮点栈式寄存器。非等待形式的这条指令（fninit）还会让
    # 协处理器终止执行当前正在执行的任何先前的算术操作。fstsw 指令取协处理器的状态字。如果系
    # 统中存在协处理器的话，那么在执行了 fninit 指令后其状态字低字节肯定为 0。

54  check_x87:
55          fninit                      # 向协处理器发出初始化命令。
56          fstsw %ax                   # 取协处理器状态字到 ax 寄存器中。
57          cmpb $0,%al                 # 初始化后状态字应该为 0，否则说明协处理器不存在。
58          je 1f                       /* no coprocessor: have to set bits */
59          movl %cr0,%eax              # 如果存在则向前跳转到标号 1 处，否则改写 cr0。
60          xorl $6,%eax                /* reset MP, set EM */
61          movl %eax,%cr0
62          ret

    # 下面是一汇编语言指示符。其含义是指存储边界对齐调整。"2"表示把随后的代码或数据的偏移位置
    # 调整到地址值最后 2 比特位为零的位置（2^2），即按 4 字节方式对齐内存地址。不过现在 GNU as
    # 直接时写出对齐的值而非 2 的次方值了。使用该指示符的目的是为了提高 32 位 CPU 访问内存中代码
    # 或数据的速度和效率。参见程序后的详细说明。
    # 下面的两个字节值是 80287 协处理器指令 fsetpm 的机器码。其作用是把 80287 设置为保护模式。
    # 80387 无需该指令，并且将会把该指令看作是空操作。

63  .align 2
64  1:      .byte 0xDB,0xE4             /* fsetpm for 287, ignored by 387 */  # 287 协处理器码。
65          ret
66
67  /*
68   *  setup_idt
69   *
70   *  sets up a idt with 256 entries pointing to
71   *  ignore_int, interrupt gates. It then loads
72   *  idt. Everything that wants to install itself
73   *  in the idt-table may do so themselves. Interrupts
74   *  are enabled elsewhere, when we can be relatively
75   *  sure everything is ok. This routine will be over-
76   *  written by the page tables.
77   */
    /*
     * 下面这段是设置中断描述符表子程序 setup_idt
     *
     * 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
     * 描述符表寄存器(用 lidt 指令)。真正实用的中断门以后再安装。当我们在其他地方认为一切
     * 都正常时再开启中断。该子程序将会被页表覆盖掉。
```

```
 */
# 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
# (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。
# 这段代码首先在 edx、eax 中组合设置出 8 字节默认的中断描述符值，然后在 idt 表每一项中
# 都放置该描述符，共 256 项。eax 含有描述符低 4 字节，edx 含有高 4 字节。内核在随后的初始
# 化过程中会替换安装那些真正实用的中断描述符项。

78 setup_idt:
79         lea ignore_int,%edx         # 将 ignore_int 的有效地址（偏移值）值➔edx 寄存器
80         movl $0x00080000,%eax       # 将选择符 0x0008 置入 eax 的高 16 位中。
81         movw %dx,%ax                /* selector = 0x0008 = cs */
                                       # 偏移值的低 16 位置入 eax 的低 16 位中。此时 eax 含有
                                       # 门描述符低 4 字节的值。
82         movw $0x8E00,%dx            /* interrupt gate - dpl=0, present */
83                                     # 此时 edx 含有门描述符高 4 字节的值。
84         lea _idt,%edi               # _idt 是中断描述符表的地址。
85         mov $256,%ecx
86 rp_sidt:
87         movl %eax,(%edi)            # 将哑中断门描述符存入表中。
88         movl %edx,4(%edi)           # eax 内容放到 edi+4 所指内存位置处。
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 */
    /*
     * 设置全局描述符表项 setup_gdt
     * 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
     * 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☺。
     * 该子程序将被页表覆盖掉。
     */
105 setup_gdt:
106        lgdt gdt_descr              # 加载全局描述符表寄存器(内容已设置好，见 234-238 行)。
107        ret
108
109 /*
110  * I put the kernel page tables right after the page directory,
111  * using 4 of them to span 16 Mb of physical memory. People with
112  * more than 16MB will have to expand this.
113  */
    /* 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。
     * 如果你有多于 16 Mb 的内存，就需要在这里进行扩充修改。
```

```
*/
# 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
# 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。
# 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)、
# 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
# 页框地址，用于指出一页内存的物理起始地址。

.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）。
pg0:

.org 0x2000
pg1:

.org 0x3000
pg2:

.org 0x4000
pg3:

.org 0x5000         # 定义下面的内存数据块从偏移 0x5000 处开始。
/*
 * tmp_floppy_area is used by the floppy-driver when DMA cannot
 * reach to a buffer-block. It needs to be aligned, so that it isn't
 * on a 64kB border.
 */
/* 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
 * 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。
 */
_tmp_floppy_area:
        .fill 1024,1,0              # 共保留 1024 项，每项 1 字节，填充数值 0。

# 下面这几个入栈操作用于为跳转到 init/main.c 中的 main()函数作准备工作。第 139 行上
# 的指令在栈中压入了返回地址，而第 140 行则压入了 main()函数代码的地址。当 head.s
# 最后在第 218 行执行 ret 指令时就会弹出 main()的地址，并把控制权转移到 init/main.c
# 程序中。参见第 3 章中有关 C 函数调用机制的说明。
# 前面 3 个入栈 0 值应该分别表示 envp、argv 指针和 argc 的值，但 main()没有用到。
# 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
# 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。
# 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
# 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 程序了。
# 有关 C 函数调用机制请参见程序后的说明。
after_page_tables:
        pushl $0                    # These are the parameters to main :-)
        pushl $0                    # 这些是调用 main 程序的参数（指 init/main.c）。
        pushl $0                    # 其中的'$'符号表示这是一个立即操作数。
        pushl $L6                   # return address for main, if it decides to.
        pushl $_main                # '_main'是编译程序对 main 的内部表示方法。
        jmp setup_paging            # 跳转至第 198 行。
L6:
        jmp L6                      # main should never return here, but
                                    # just in case, we know what happens.
                                    # main 程序绝对不应该返回到这里。不过为了以防万一，
                                    # 所以添加了该语句。这样我们就知道发生什么问题了。
```

```
145
146 /* This is the default interrupt "handler" :-) */
    /* 下面是默认的中断“向量句柄”☺ */
147 int_msg:
148         .asciz "Unknown interrupt\n\r"      # 定义字符串“未知中断(回车换行)”。
149 .align 2                   # 按 4 字节方式对齐内存地址。
150 ignore_int:
151         pushl %eax
152         pushl %ecx
153         pushl %edx
154         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
155         push %es            # 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。
156         push %fs
157         movl $0x10,%eax     # 置段选择符（使 ds,es,fs 指向 gdt 表中的数据段）。
158         mov %ax,%ds
159         mov %ax,%es
160         mov %ax,%fs
161         pushl $int_msg      # 把调用 printk 函数的参数指针（地址）入栈。注意！若 int_msg
162         call _printk        # 前不加'$'，则表示把 int_msg 符号处的长字（'Unkn'）入栈☺。
163         popl %eax           # 该函数在/kernel/printk.c 中。'_printk'是 printk 编译后模块中
164         pop %fs             # 的内部表示法。
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“普通”函数仅使用低于 1Mb 的地址空间，或者是使用局部数据
 * 空间，该地址空间将被映射到其他一些地方去 -- 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☹）。
 */
# 上面英文注释第 2 段的含义是指在机器物理内存中大于 1MB 的内存空间主要被用于主内存区。
# 主内存区空间由 mm 模块管理。它涉及到页面映射操作。内核中所有其他函数就是这里指的一般
#（普通）函数。若要使用主内存区的页面，就需要使用 get_free_page()等函数获取。因为主内
# 存区中内存页面是共享资源，必须有程序进行统一管理以避免资源争用和竞争。
#
# 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 页页表则属于内核专用，它们一一映射线性地址起始 16MB 空间范围到物理内存上。对于
# 新的进程，系统会在主内存区为其申请页面存放页表。另外，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               # eax 内容存到 es:edi 所指内存位置处，且 edi 增 4。

    # 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。
    # 例如"$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 $0xfff007,%eax         /*  16Mb - 4096 + 7 (r/w user,p) */
                                        # 最后 1 项对应物理内存页面的地址是 0xfff000，
                                        # 加上属性标志 7，即为 0xfff007。
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。
    # 设置页目录表基址寄存器 cr3 的值，指向页目录表。cr3 中保存的是页目录表的物理地址。
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 140 行压入堆栈中的 main 程序的地址弹出，并跳转到/init/main.c
    # 程序去运行。本程序到此就真正结束了。

219
220 .align 2                           # 按 4 字节方式对齐内存地址边界。
221 .word 0                           # 这里先空出 2 字节，这样 224 行上的长字是 4 字节对齐的。

    ! 下面是加载中断描述符表寄存器 idtr 的指令 lidt 要求的 6 字节操作数。前 2 字节是 idt 表的限长，
    ! 后 4 字节是 idt 表在线性地址空间中的 32 位基地址。
222 idt_descr:
223         .word 256*8-1               # idt contains 256 entries  # 共 256 项，限长=长度 - 1。
224         .long _idt
225 .align 2
226 .word 0

    ! 下面加载全局描述符表寄存器 gdtr 的指令 lgdt 要求的 6 字节操作数。前 2 字节是 gdt 表的限长，
    ! 后 4 字节是 gdt 表的线性基地址。这里全局表长度设置为 2KB 字节（0x7ff 即可），因为每 8 字节
    ! 组成一个描述符项，所以表中共可有 256 项。符号_gdt 是全局表在本程序中的偏移位置，见 234 行。
227 gdt_descr:
228         .word 256*8-1               # so does gdt (not that that's any   # 注：not → note
229         .long _gdt                  # magic number, but it works for me :^)
230
231         .align 3                    # 按 8（2^3）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、系统调用段描述符，其中
    # 系统调用段描述符并没有派用处，Linus 当时可能曾想把系统调用代码专门放在这个独立的段中。
    # 后面还预留了 252 项的空间，用于放置所创建任务的局部描述符(LDT)和对应的任务状态段 TSS
    # 的描述符。
    # (0-nul, 1-cs, 2-ds, 3-syscall, 4-TSS0, 5-LDT0, 6-TSS1, 7-LDT1, 8-TSS2 etc...)

234 _gdt:   .quad 0x0000000000000000        /* NULL descriptor */
235         .quad 0x00c09a0000000fff        /* 16Mb */      # 0x08，内核代码段最大长度 16MB。
236         .quad 0x00c0920000000fff        /* 16Mb */      # 0x10，内核数据段最大长度 16MB。
237         .quad 0x0000000000000000        /* TEMPORARY - don't use */
238         .fill 252,8,0                   /* space for LDT's and TSS's etc */  # 预留空间。
```