保护模式

倪小凡

2016-04-06

8088，8086

- 实模式下，段在内存中固定的位置（物理地址 = 段值*16 + 偏移 ）；通过改变段寄存器的值，我们可以随心所欲的访问内存任何一个单元；而丝毫不受到限制，不能对内存访问加以限制，也就谈不上对系统的保护。因此，在实模式下是无法构造现代意义的操作系统的。
- 内存中每个字节的地址能由不止一个的段基址加偏移表示，比如04808能够由047C:0048, 047D:0038,047E:0028或047B:0058表示，分段地址之间的比较将复杂化。

80286

- 16位的保护模式下，段不在固定的位置，甚至可以不在内存中（虚拟内存技术），内存中存储的只是当前程序运行需要的指令和数据，为了向前兼容，仍然使用SEG:OFFSET这样来表示，只不过这是得到的这个值不是实际物理地址，而是变成一个索引，这个索引指向一个数据结构的一个表项，表项中详细定义了是否在内存中，段的起始地址、界限、属性等内容，这就是GDT(也可能是LDT)，即“描述符”， 段大小还是限制在64k。

80386

- 32位的保护模式，段大小扩展到4GB，段可以被划分为更小的单位，4K（分页），虚拟内存现在以分页的方式工作，原先80286中一整段要么在内存中，要么不再内存中，现在在内存中的可能只是段的一部分。

保护模式（粗略）

- 比较粗略，只是建立对保护模式的印象。
- 当一条访问内存指令发出一个内存地址时，CPU是这样来归纳出实际上应该放在数据总线的地址：
  1. 根据指令的性质来确定使用哪个段寄存器，例如，放在转移指令中的地址在代码段，而取数据的指令中的地址在数据段；
  2. 根据段寄存器的内容，找到相应的“描述符”；
  3. 从描述符中得到基地址；
  4. 将指令中发出的地址作为位移，与描述符中界限相比，看是否越界；
  5. 将指令的性质与描述符中的访问权限来确定是否越权；
  6. 将指令中发出的地址作为位移，与基地址相加得出实际的“物理地址”。

GDT/LDT

- GDT，Global Descriptor Table,是一张存放Descriptor的表，可在全局内访问,所有进程想要访问全局可见的段时，从GDT查询，有且只有一个。进程从GDTR寄存器中获得GDT的位置，向它发起查询。

- LDT(Local)与GDT相同，但是不是全局的，对于某个进程，它只知道它自己的LDT。每个进程有自己的LDT，访问自己的段时从LDT查询。进程从LDTR寄存器中获得LDT的位置，向它发起查询。

一堆疑惑

- Descriptor是什么？有什么作用？怎样才能得到Descriptor？

- GDTR/LDTR寄存器存储的是GDT/LDT在内存中的位置，但是只知道他们的位置怎么能取到Descriptor呢？因为还缺少Descriptor在表中的具体的偏移。

- 下面慢慢解释……

Descriptor

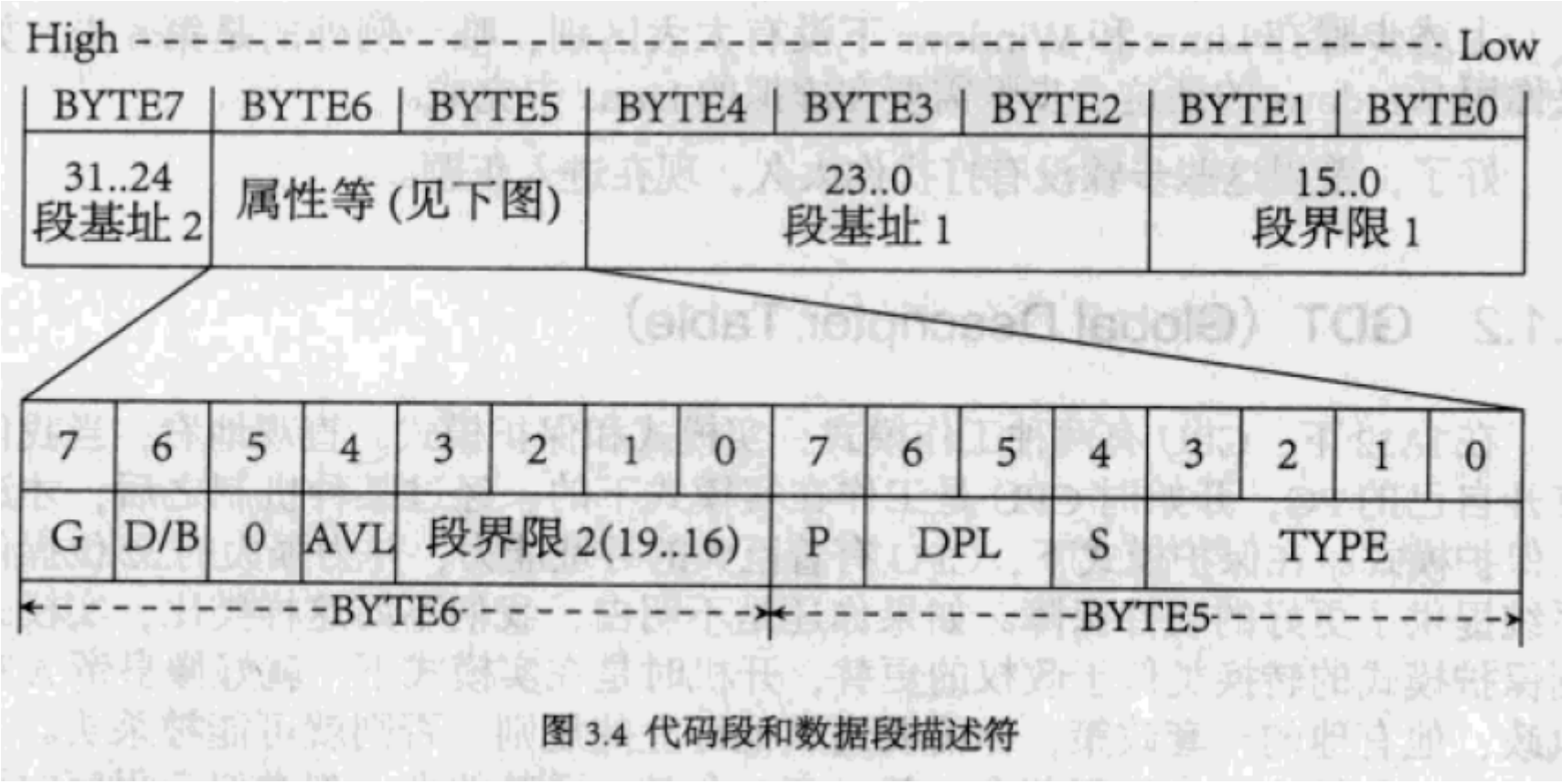


图 3.4 代码段和数据段描述符

Descriptor

- 保护模式下引入描述符来描述各种数据段，所有的描述符均为8个字节（0-7），由第5个字节说明描述符的类型。类型不同，描述符的结构也有所不同。
- 若干个描述符集中在一起组成描述符表，而描述符表本身也是一种数据段，也使用描述符进行描述。从现在起，“地址转换”由描述符表来完成，从这个意义上说，描述符表是一张地址转换函数表。

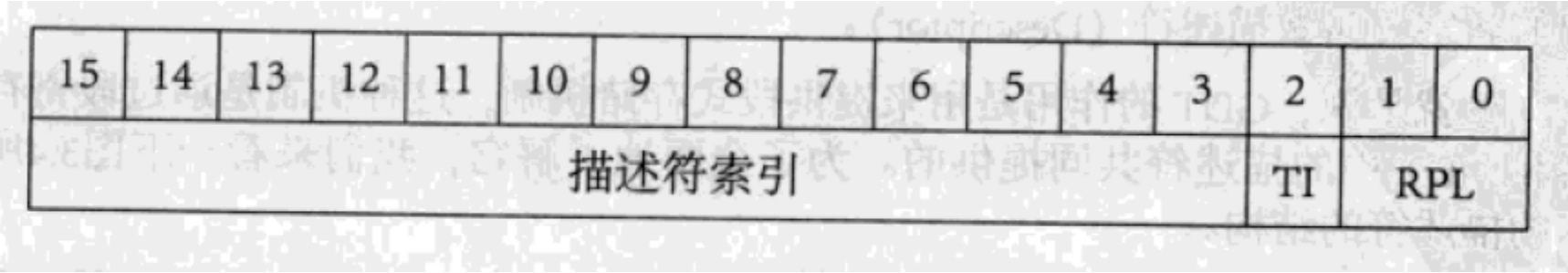
Selector

- GDT中的每一个描述符都定义了一个段，那么cs，ds等段寄存器是如何和这些段对应起来的呢(即如何通过段寄存器找到对应的描述符呢)？

```
mov     ax, SelectorVideo
mov     gs, ax
```

- 段寄存器gs的值变成了SelectorVideo；
- 不难推测到选择子的大致结构，肯定包含了描述符在描述符表中的位置（也就是之前说的偏移）。

Selector



- 选择子是一个2字节的数，共16位，最低2位表示RPL（请求特权等级），第3位表示查表是利用GDT（全局描述符表）还是LDT（局部描述符表）进行，最高13位给出了 所需的描述符在描述符表中的地址。（注：13位正好足够寻址8K项）有了以上三个概念之后可以进一步工作了，现在程序的运行与实模式下完全一样。各段寄存器仍然给出一个“段值”，只是这个“假段值”到真正的段地址的转换不再是“左移4位”，而是利用描述符表来完成。

GDTR/LDTR

- 已经知道了描述符在描述符表中的位置，那描述符表的位置又由什么来指示呢？
  - 为了解决这个问题，显然需要引入新的寄存器用于指示GDT/LDT在内存中的位置。
  - 在80x86系列中引入了两个新寄存器GDTR和LDTR。

GDTR

- GDTR用于表示GDT在内存中的段地址和段限（就是表的大小），因此GDTR是一个48位的寄存器，其中32位表示段地址，16位表示段限（最大 64K，每个描述符8字节，故最多有64K/8=8K个描述符）。

- 刚才所说的Selector，它的高13位表示的就是描述符在描述符表中的位置，和这里刚好对应。

LDTR

- LDTR用于表示LDT在内存中的位置，但是因为LDT本身也是一种数据段，它必须有一 个描述符，且该描述符必须放在GDT中，因此LDTR使用了与DS、ES、CS等相同的机制，其中只存放一个“选择子”，通过查GDT表获得LDT的真正内存地址。

- 为什么LDT要放在GDT中？

LDTR

- 除了选择子的bit3一个为0一个为1用于区分该描述符是在GDT中还是在LDT中外，描述符本身的结构完全一样。

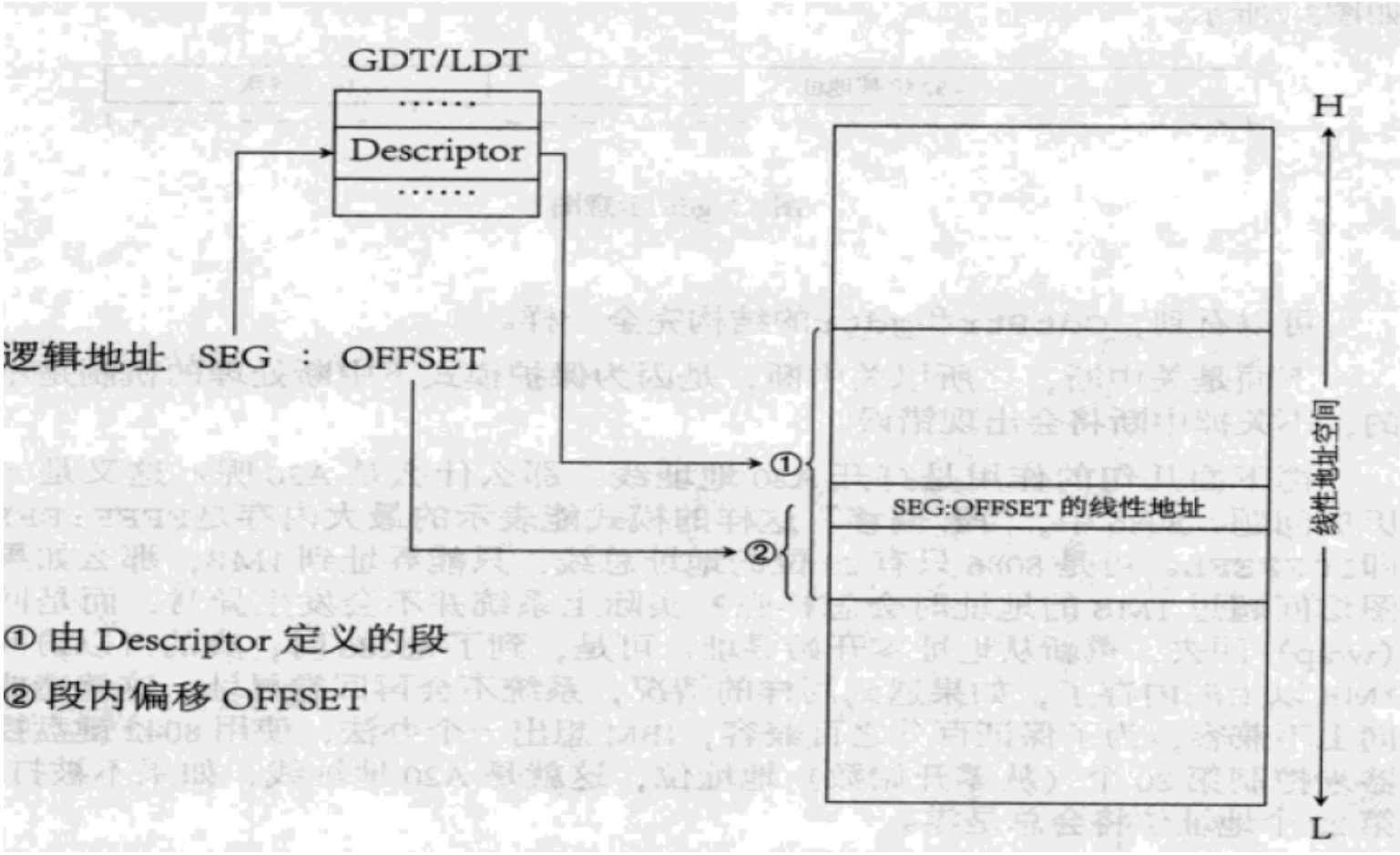
- 既然是这样，为什么要将LDT放在GDT中而不是像GDT那样找一个GDTR寄存器呢？

  - GDT表只有一个，是固定的；而LDT表每个任务就可以有一个，因此有多个，并且由于任务的个数在不断变化其数量也在不断变化。
  - 如果只有一个LDTR寄存器显然不能满足多个LDT的要求。因此intel的做法是把它放在放在GDT中。

引申—IDT(中断描述符表)

- 在80x86系列中为中断服务提供中断/陷阱描述符，这些描述符构成中断描述符表（IDT），并引入一个48位的全地址寄存器存放 IDT 的内存地址。理论上IDT表同样可以有8K项，可是因为80x86只支持256个中断，因此IDT实际上最大只能有256项（2K大小）。

保护模式（回顾）


GDT/LDT
Descriptor
逻辑地址 SEG ： OFFSET
①
②
SEG:OFFSET 的线性地址
H
线性地址空间
L
① 由 Descriptor 定义的段
② 段内偏移 OFFSET

保护模式（回顾）

- 保护模式下具体一点的寻址过程：

1. 根据指令的性质来确定使用哪个段寄存器，例如，放在转移指令中的地址在代码段，而取数据的指令中的地址在数据段；
2. 根据段寄存器的内容(选择子)，首先判断描述符是在GDT中还是在LDT中，如果是在GDT中，根据GDTR以及该段寄存器的内容找到相应的“描述符”；如果是在LDT中，根据LDTR（选择子）以及GDTR的内容找到LDT的描述符，得到LDT的地址，然后再根据段寄存器内容找到相应的“描述符”；
3. 从描述符中得到基地址；
4. 将指令中发出的地址作为位移，与描述符中界限相比，看是否越界；
5. 将指令的性质与描述符中的访问权限来确定是否越权；
6. 将指令中发出的地址作为位移，与基地址相加得出实际的“物理地址”。

THX.