保护模式内存管理-读书笔记

2021113117 王宇轩

1. 内存管理概览

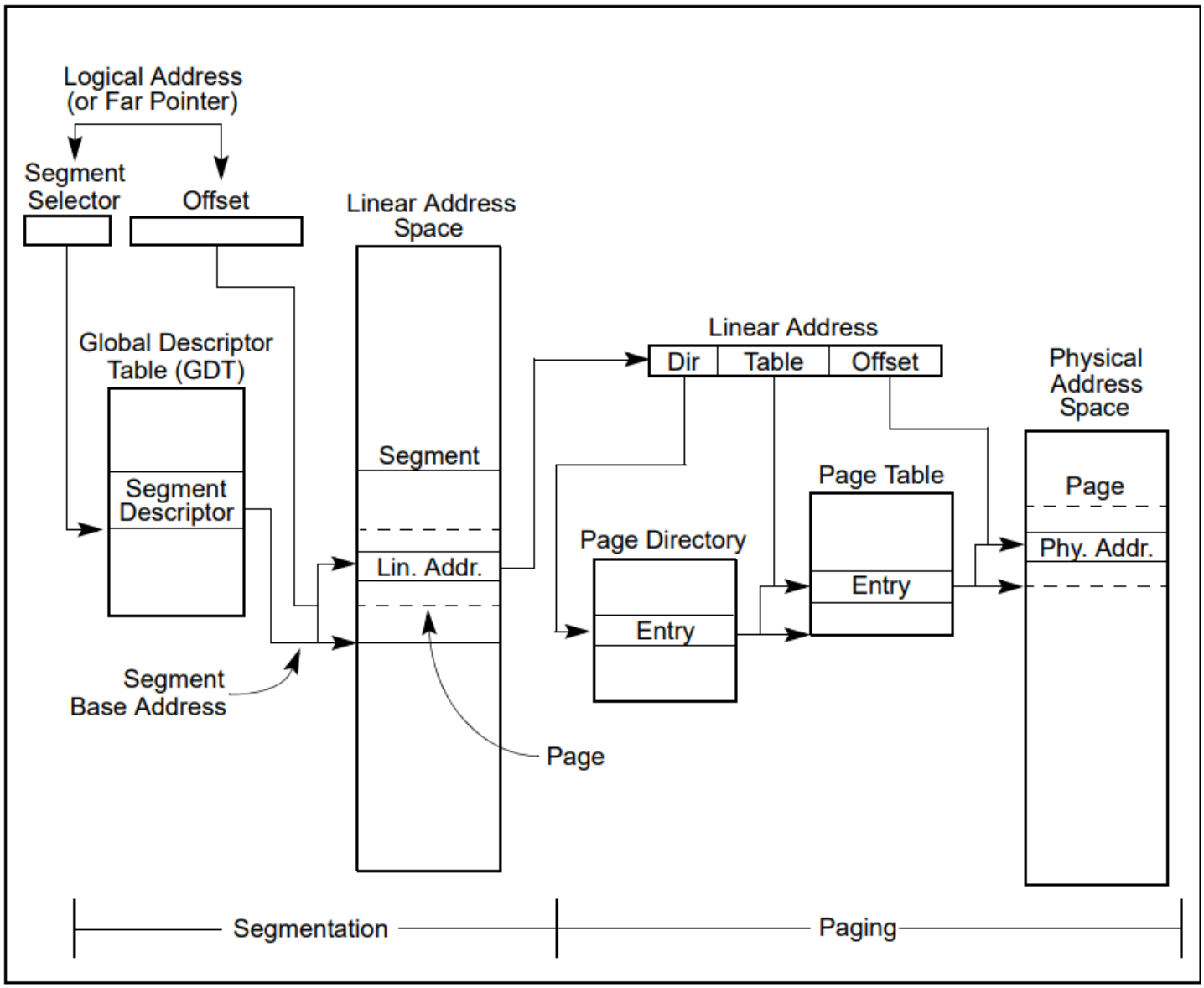

Figure 3-1. Segmentation and Paging

- IA-32架构中内存管理工具分为两部分：**分段**和**分页**。两种机制都可以配置支持单程序（单任务）、多任务或共享内存的多处理器系统。
- 保护模式下必须启用某种形式的分段，没有模式位来关闭分段，但分页是可选的。

有关**分段**：

- 提供了一种隔离单个代码、数据和堆栈模块的机制，以便多个程序（或任务）可以在同一个处理器上运行而不会相互干扰。

- 将处理器可寻址的空间（称**线性地址（Linear Address）空间**）分为**段**，可装载程序的代码、数据和栈，或是系统数据结构（如TSS、LDT）。
- 处理器执行多程序/多任务时，每个程序可分得独有的段集合，处理器会实行段间隔离并保障程序之间不会因写入到其他程序的段而相互干扰执行。
- 还允许段的分类，可以限制对特定类型的段执行的操作。
- 系统中的所有段都包含在处理器的**线性地址空间**中，提供**逻辑地址**（**Logical Address**，亦称远指针，包含一个**段选择子**和一个**偏移量**）来定位特定段种的字节。
- **段选择子**是段的唯一标识，其中会有一个偏移量用于某个描述符表（GDT等）来定位一个**段描述符**。段描述符是一个数据结构，每个段都有，指定了段的大小、访问权限、特权等级、类型、基地址（首字节在线性地址空间中的位置）。基地址+逻辑地址中的偏移量=段中字节的位置（就是处理器线性地址空间中的**线性地址**）。

有关**分页**：

- 提供了一种实现传统的按需分页的虚拟内存系统机制，程序执行环境的各个部分按需映射到物理内存中。还可用于提供多个任务之间的隔离。因多任务计算系统所需的线性地址空间远大于经济上可行的物理存储大小，分页机制提供了一种线性地址的“**虚拟化**”。
- 如果不使用分页，则将处理器的线性地址空间直接映射到处理器的**物理地址（Physical Address）空间**处理器可以在其地址总线上生成的地址范围。
- 分页支持了一种由少量的物理内存（RAM或ROM）和一些磁盘存储模拟出的大线性地址空间——**虚拟内存**。启用分页时，每个段被分为若干页（通常每个页大小为4 KByte），每个页都存在物理内存或磁盘中，操作系统或executive[1]维护一个**页目录**和一组**页表**来跟踪这些页面，当程序或任务访问线性地址空间中的某个位置时，处理器会用页目录和页表来实现从线性地址到物理地址的变换。
- 若访问的页面不在物理内存中，处理器生成一个缺页异常，操作系统和executive将目标页读入内存并继续执行原程序。
- 分页正常实行时，页面在物理内存和磁盘中的调度对正常执行的程序是**透明的**，即使为16位IA-32的处理器编写的程序也可以在虚拟8086模式下执行时启用分页。

Q&A

1. **什么是“executive”?**
   可理解为操作系统内核中负责维护和管理页目录和页表的组件或模块。

> Windows并没有将运行在Ring 0的代码全部视为内核，而是区分为Kernel和Executive，Executive可以理解为“管理层”的意思，解释为“执行体”不合理。其中，Kernel是狭义的内核，里面的代码包括用到的数据，都是常驻在物理内存中的，不支持分页机制。而除此之外的代码和数据，是支持分页机制的，并且可以被交换到pagefile中，即并非总是在物理内存中的。对于驱动来说，应该属于后者，因此在驱动中的函数的头部都会使用PAGED_CODE来判断一下.

2. 分段机制

- 分段机制可用于实现各种系统设计。
- Intel 64架构的**IA-32e**模式中的分段：**兼容模式**下，分段的功能就像使用传统的16位或32位保护模式语义一样。**64位模式**下，通常（但并不完全）不启用分段，创造了一个扁平的64位线性地址空间。处理器将CS，DS，ES，SS[1]的段基视为0，由此线性地址与有效地址等效。FS和GS段是例外，它们(保存段基)可以在线性地址计算中用作额外的基寄存器，有助于寻址本地数据和某些操作系统数据结构。**在64位模式下，处理器不会在运行时执行段限制检查。**
- 任意下述分段模式都可启用分页，将（段所映射到的）线性地址空间分为页，如图3-1，这些**线性地址页**又被映射到物理地址空间中的页。分页机制提供了几个页级保护设施，可以与段保护设施一起使用，也可以代替段保护设施使用。例如，它允许在逐页的基础上实施读写保护。分页机制还提供两级用户-主管保护，也可以逐页指定。

Q&A

1. **段寄存器缩写：**
   CS——Code Segment，DS——Data Segment，ES——Extra Segment，SS——Stack Segment，FS和GS寄存器是额外的段寄存器（Additional Segment）。

2.1 基础扁平模型（Basic Flat Model）

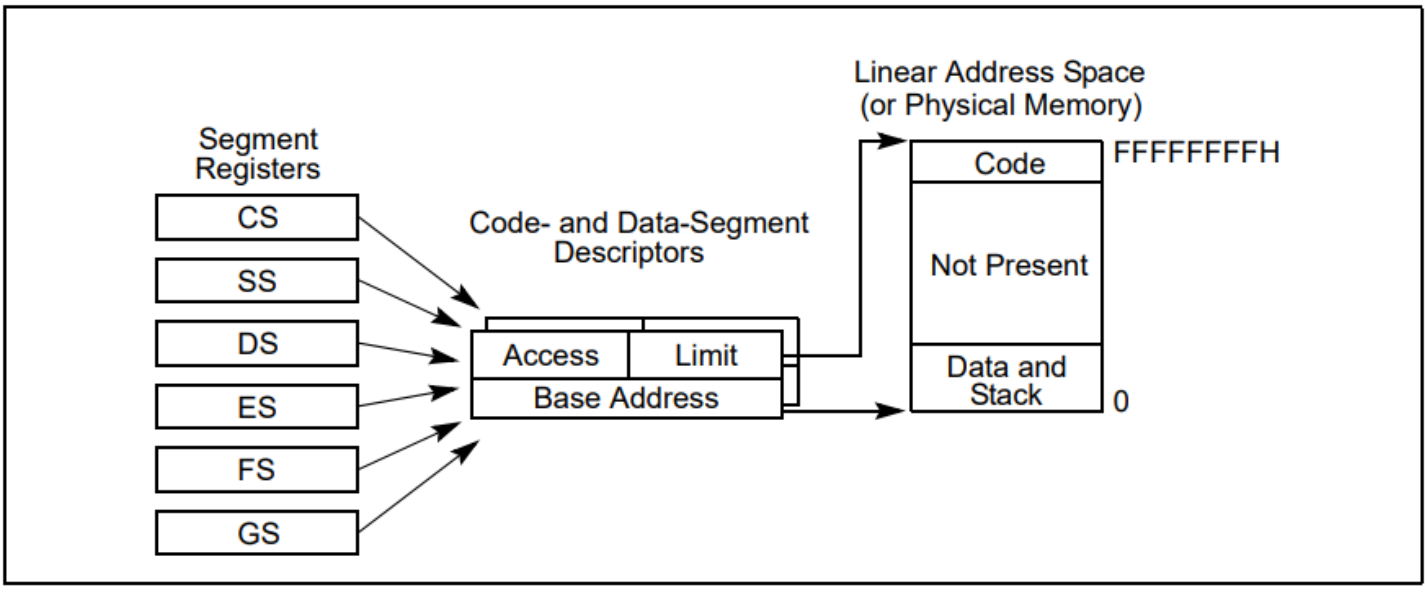

Figure 3-2. Flat Model

- 最简单的内存模型，操作系统和应用程序可访问一个**连续的、未分段的**地址空间。这种模型对系统设计者和应用程序员隐藏了分段机制。
- 要使用 IA-32 架构实现基本的平面内存模型，必须至少创建两个段描述子，一个用于引用代码段，一个用于引用数据段，**这两个段都被映射到整个线性地址空间**，即两个段描述子具有相同的基地址值0和相同的4GB的段限制。这样设置段限制，即使某地址没有物理内存也不会出现超出限制的内存引用异常。ROM（EPROM）一般位于物理地址空间的顶部，因为处理器从

FFFF_FFF0H开始执行。 RAM（DRAM）一般放在地址空间的底部，因为复位初始化后DS数据段的初始基地址为0。

2.2 受保护的扁平模型（Protected Flat Model）

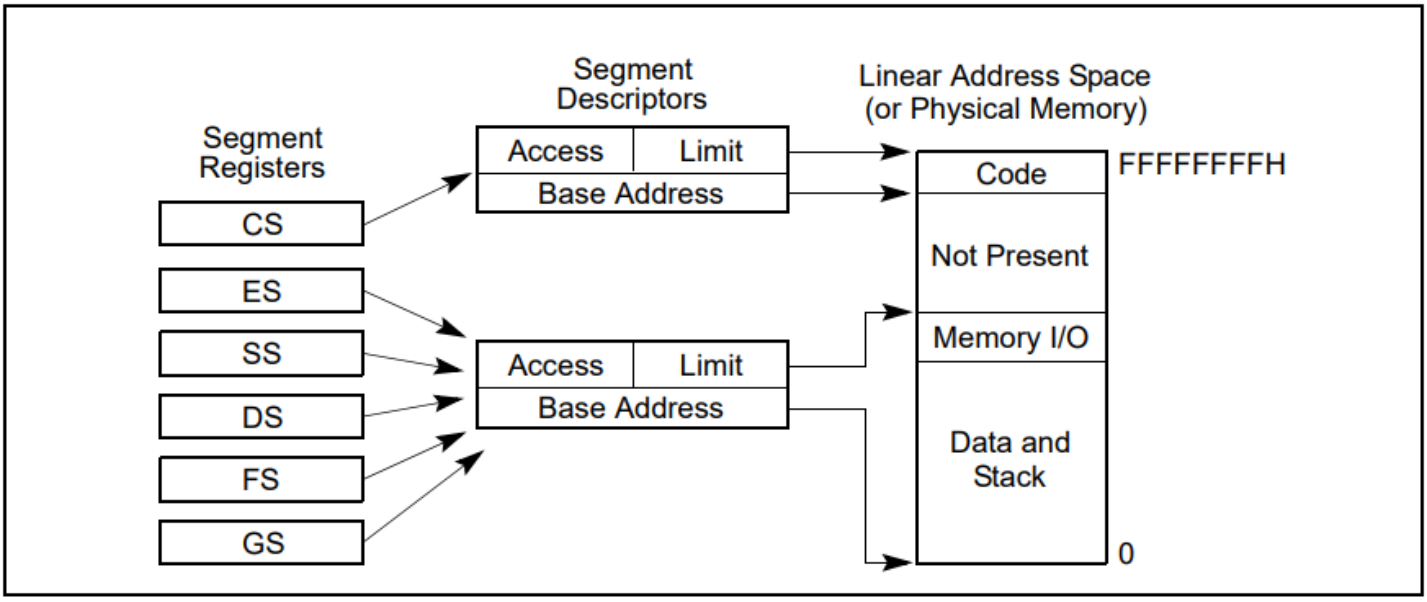

Figure 3-3. Protected Flat Model

- 与基础扁平模型类似，除了**段限制设置为仅包括物理内存实际存在的地址范围**。出现试图访问不存在的内存时会生成一个**#GP**（general-protection）异常，以此提供一个对于程序bug的最低程度硬件保护。
- 可以进一步复杂化。如，对于隔离了用户和主管的数据、代码的分页机制，需要四个段：用户的代码段数据段，特权等级3；主管的代码段数据段，特权等级0。通常，这些段从线性地址空间0处开始互相覆盖。
- 带有简单分页结构的此分段模型可以保护操作系统不受应用程序的影响，并且，如果增加每个任务或程序单独的分页结构，可以避免程序们互相影响。一些流行的多任务操作系统也采用了类似的设计。

2.3 多段模型（Multi-Segment Model）

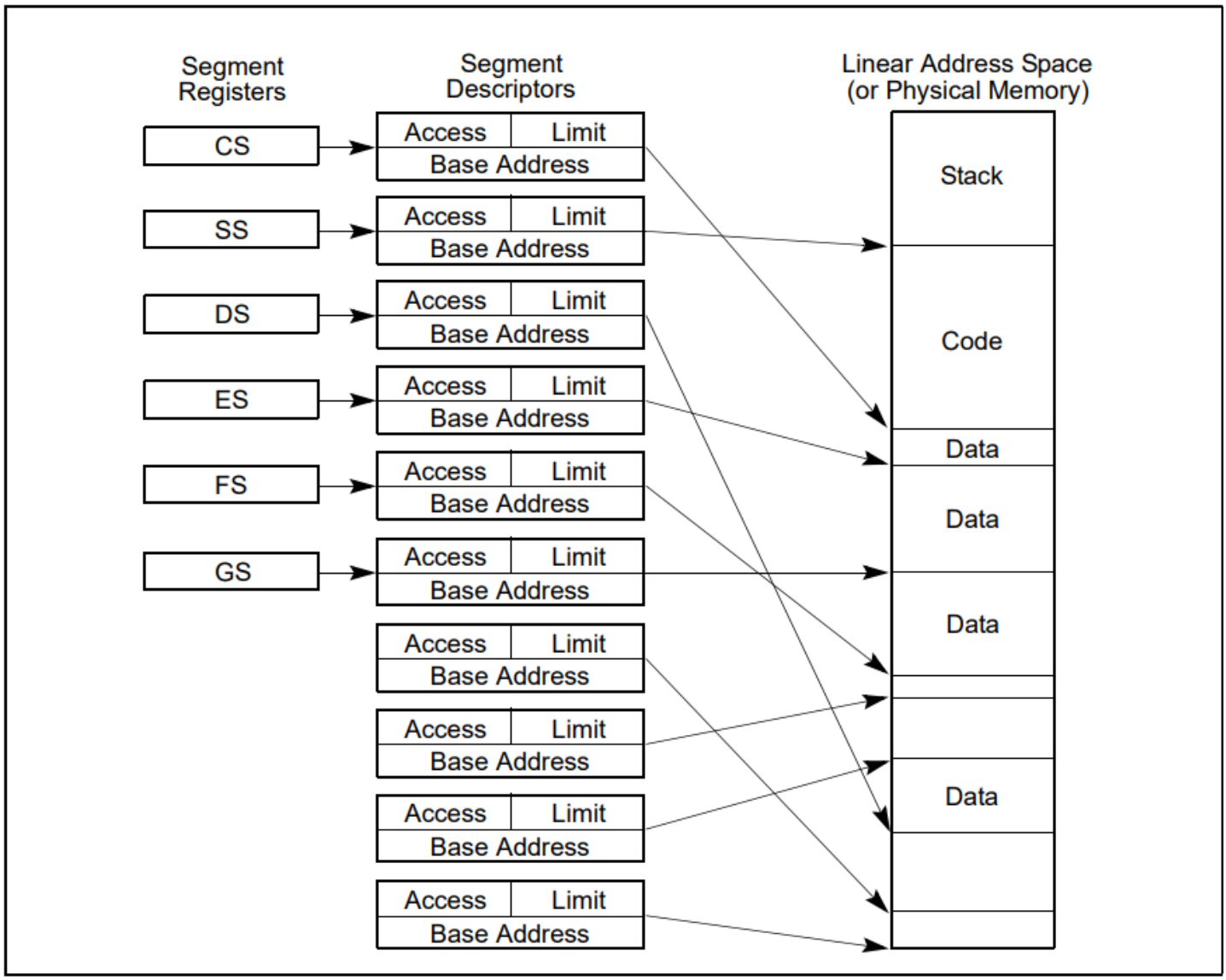

Figure 3-4. Multi-Segment Model

- 充分运用分段机制的能力，为代码、数据结构、程序和任务提供硬件强制保护。**每个程序或任务都有各自的段描述符表和段**，这些段可以私有，也可在程序间共享。对所有段和系统上运行的各个程序的执行环境的访问由硬件控制。
- **访问检查**不仅可以用来防止引用段限制之外的地址，还可以用来防止在某些段中执行不允许的操作。例如，有些代码段被指定为只读段，可以使用硬件实现防止写入代码段的功能。为段创建的访问权限信息也可用于设置**保护环（protection rings）**或**保护级别**。保护级别可用于保护操作系统执行过程中免受应用程序未经授权的访问。

3. 逻辑地址和线性地址的转换

3.1 段选择子（Segment Selectors）

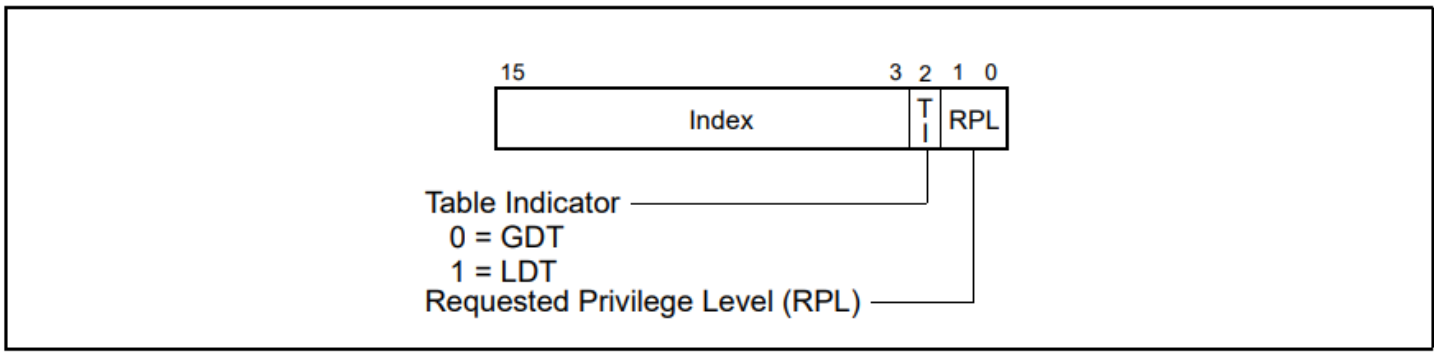

Figure 3-6. Segment Selector

段选择子是段的16位标识符，并不直接指向段，而是**指向定义段的段描述符**。包含：

- **索引**：即图中Index，选择GDT或LDT中的8192个描述符之一。处理器将索引值乘以8（段描述符中的字节数），并将结果与到GDT或LDT的基地址（从GDTR或LDTR寄存器获取）相加。
- **TI标志**：选择要使用的描述符表：如图，0=GDT，1=LDT。
- **RPL**：指定**选择子优先级**，从0到3特权等级由高至低。

处理器不使用GDT的第一个条目，指向此条目的段选择子（Index为0，TI=0）作为“**空段选择子**”使用。当段寄存器（CS或SS除外）加载空选择子时，处理器不会产生异常。但当使用包含空选择子的段寄存器访问内存时，会产生异常。空选择子可用于初始化未使用的段寄存器。使用空段选择子加载CS或SS寄存器会导致生成一般保护异常#GP。

段选择子作为一个指针变量的一部分对应用程序可见，但选择子的值通常由链接编辑器或链接加载器分配或修改，而不是应用程序进行该操作。

3.2 段寄存器（Segment Registers）

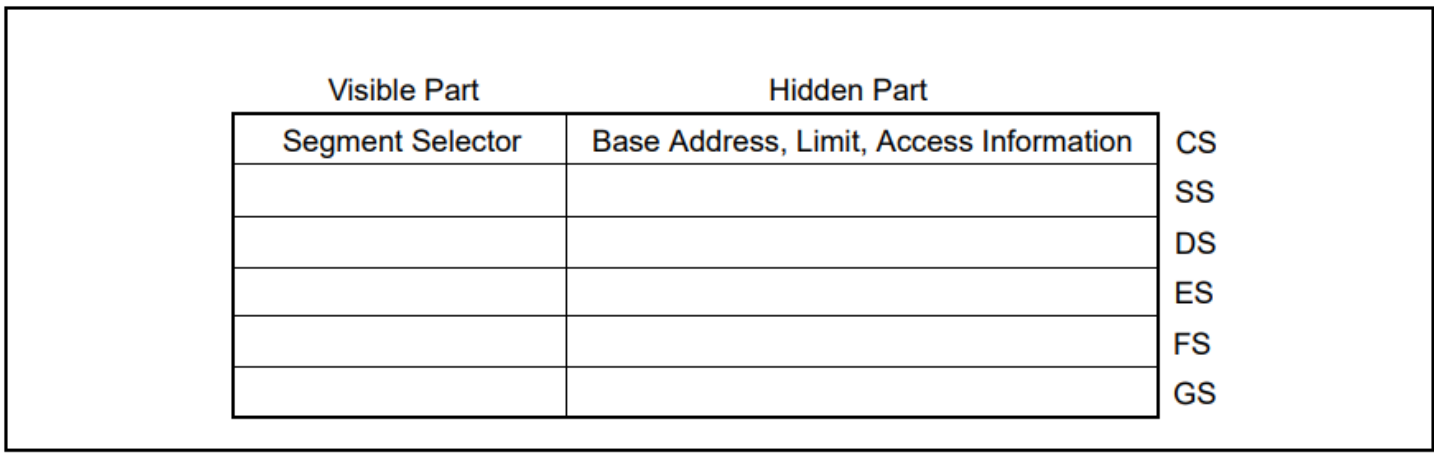

Figure 3-7. Segment Registers

为减少地址转换时间和编码复杂性，处理器提供了**最多可容纳6个段选择子的寄存器**。这些段寄存器中的每一个都支持特定类型的内存引用（代码、堆栈或数据）。要执行任何类型的程序，至少代码段寄存器（CS）、数据段寄存器（DS）和堆栈段寄存器（SS）必须加载有效的段选择子。处理器还提供了三个额外的数据段寄存器（ES、FS 和 GS），它们可用于为当前正在执行的程序（或任务）提供额外的数据段。

段的段选择子必须已经加载到其中一个段寄存器中，处理器才能完成对段程序的访问。因此，尽管一个系统可以定义数千个段，**只有6个可以立即访问执行**。程序执行期间可通过将其他段的段选择器加载到寄存器中使其可用。

每个段寄存器都有可见部分和不可见部分（不可见部分也即描述符缓存/影子寄存器）。当某个段选择子被加载进可见部分，处理器也会将对应的基地址、段限制和访问控制信息加载进隐藏部分。缓存在段寄存器中的信息（包含可见的和隐藏的）**允许处理器对逻辑地址直接进行转换**，而无需花费额外的总线周期从段描述符中读取基地址和段限制。在多个处理器可以访问相同段描述符表的系统中，当表被修改时，由软件负责重新加载段寄存器。若是不这样做，缓存在段寄存器中的没被修改过的段描述子在修改操作仍可能会被使用。

处理器可以使用两种加载指令完成对**段寄存器的加载**：

1. 直接加载指令，例如 `MOV` 、 `POP` 、 `LDS` 、 `LES` 、 `LSS` 、 `LGS` 和 `LFS` 指令。这些指令显式地引用段寄存器。 `MOV` 指令也可用于将段寄存器的可见部分存储到通用寄存器中。
2. 隐含的加载指令，例如 `CALL` 、 `JMP` 和 `RET` 指令的远指针版本， `SYSENTER` 和 `SYSEXIT` 指令，以及 `IRET` 、 `INTn` 、 `INTO` 和 `INT3` 指令。这些指令改变CS寄存器（有时也改变其他段寄存器）的内容，作为它们操作的附带部分。

3.3 段描述符（Segment Descriptors）

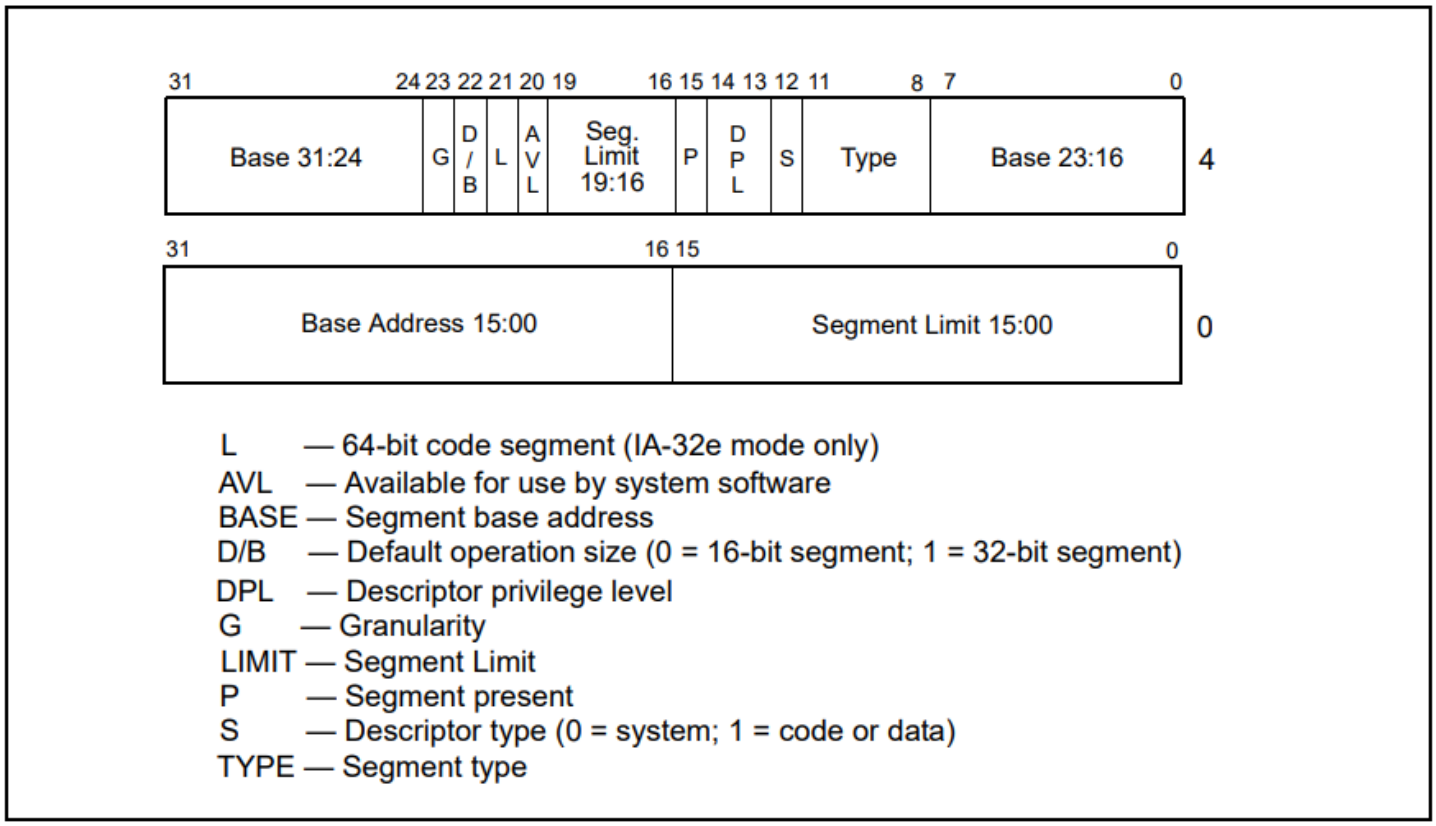

Figure 3-8. Segment Descriptor

段描述子是GDT或LDT中的一种数据结构，为处理器提供段的大小和位置，以及访问控制和状态信息，通常由编译器、链接器、加载器，或操作系统或executive创建，而不是应用程序。上图为所有类型的段描述子的通用格式，其中：

- **段限制字段（Segment limit field）**：
  指定段的大小，如图，处理器将两个字段合起来形成一个20位长的值。处理器对其的解释受**G（granularity）标志**影响：
  - 若G标志为0，则段大小范围为1B到1MB，并且以1B为增量单位；
  - 若G标志为1，则段大小范围为4KB到4GB，并且以4KB为增量单位。

  处理器以两种不同的方式使用段限制，这取决于段是向上扩展段还是向下扩展段。对于**向上扩展段**，逻辑地址中偏移量的范围是0到段限制，超出段限制的偏移量会引发#GP异常（对于SS寄存器是栈错误#SS异常，stack-fault exceptions）。对于**向下扩展段**，段限制功能相反，偏移量的范围可以从段限制+1到FFFFFFFFH或FFFFH，具体取决于B标志的设置。小于或等于段限制的偏移量会产生#GP或#SS。减小向下扩展段的段限制字段的值会在段地址空间的底部（不是顶部）分配新的内存。IA-32架构的堆栈总是**向下增长**，使得这种机制便于扩展堆栈。
- **基址字段（Base address fields）**：
  指定了段的第0个字节在4GB线性地址空间中的位置。处理器将三个基地址字段放在一起以形成一个32位的值。段基地址应与16字节边界对齐。尽管16字节对齐并不是必须的要求，但对齐能使程序在16字节边界上对齐代码和数据，以此来最大限度地提高性能。
- **类型字段（type field）**：
  指示段或门类型，并指定可以对段进行的访问类型和段的增长方向。处理器对该字段的解释取决

于描述子类型标志位指定的是应用程序（代码或数据）描述子还是系统描述子。对于代码、数据和系统描述子，类型字段的编码各不相同。

- **S标志**：
  描述符类型，指定段描述符是否用于系统段（S=0）或一个代码或数据段（S=1）。
- **DPL字段**：
  指定段的优先级。优先级范围可以从0到3，其中0是最高优先级。DPL用于控制对段的访问。
- **P标志**：
  表示段是否存在于内存中（1是0否）。如果加载进段寄存器的段选择子指向的段描述符中此标志为0，处理器生成段不存在异常（**#NP**）。内存管理软件可以使用这个标志来控制在给定的时间哪些段实际加载到物理内存中。它为管理虚拟内存提供了一个分页之外的额外控制。
  P标志被清除时的段描述符格式如下，此时操作系统或executive可以随意使用标记Available的字段来存储其自己的数据，例如关于丢失段的位置的信息。

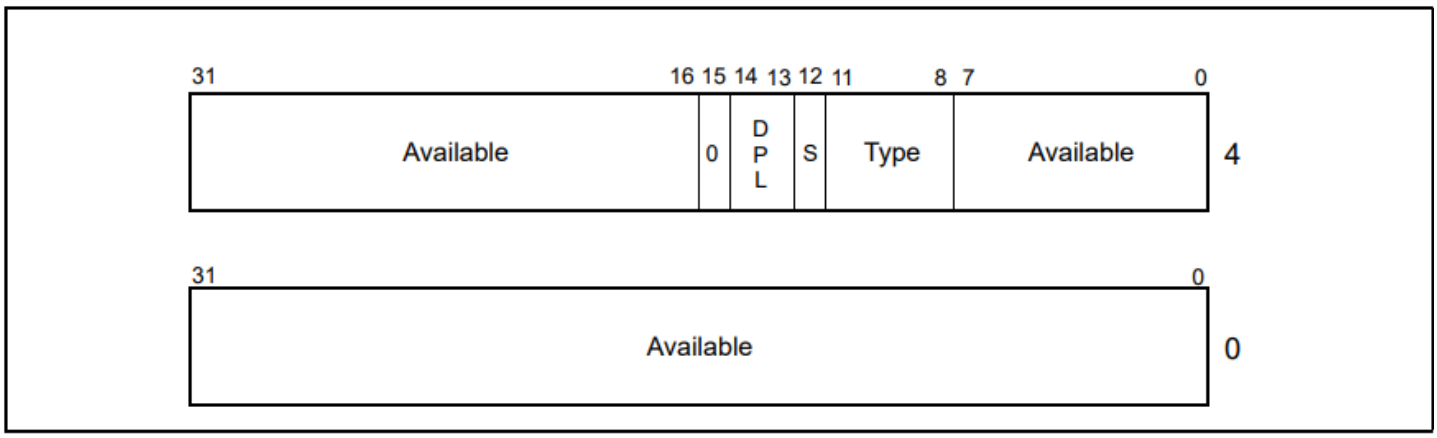

Figure 3-9. Segment Descriptor When Segment-Present Flag Is Clear

- **D/B标志**：
  根据段描述符是可执行代码段，向下扩展数据段，还是栈段，此标志有不同的功能。（对于32位代码和数据段，此标志应始终设置为1，对于16位代码和数据段，此标志应始终设置为0。）
  - **可执行代码段**：称D标志，指示段中指令引用的有效地址和操作数的默认长度。若为1，则假定32位地址和32位或8位操作数；若为0，则假定16位地址和16位或8位操作数。指令前缀66H可用于选择除默认值以外的操作数大小，前缀67H可用于选择除默认值以外的地址大小。
  - **栈段（SS寄存器指向的数据段）**：称B标志（big），指示用于隐式堆栈操作（如入栈、出栈和调用）的栈指针的大小。若为1，用32位的栈指针并存于32位的ESP寄存器；若为0，用16位栈指针并存于16位的SP寄存器。如果栈段被设置为一个向下扩展段，B标志也指定栈段的上界，如下所述。
  - **向下扩展段**：称B标志，指定段的上界，若为1，则上限为FFFFFFFFH（4GB）；若为0，则上限为FFFFH（64KB）。
- **G标志**：
  确定段限制字段的缩放比例。若为1，段限制以4KB为单位进行解释；若为0，段限制以1B（字节）为单位进行解释。（此标志不影响基地址的粒度，它始终是字节粒度。）将G标志置1后，处理器在检查偏移量时不检查偏移量的十二个最低有效位。比如将G标志置1后，又将段限制设置为0，有效的偏移量为0-4095。

- **L标志**：
  IA-32e模式下，该标志位指示代码段是否包含本机64位代码。若为1，此代码段中的指令以64位模式执行；若为0，此代码段中的指令以兼容模式执行。如果设置了L标志位，则必须清除D标志位。当不处于IA-32e模式下，或者该段并非代码段时，L标志位应始终设置为0。
- **可用（Available）位和保留（Reserved）位**：
  段描述符的第二个双字的第20位可供系统软件使用。

3.4 逻辑地址转换

保护模式中，在系统架构层面，处理器使用两个阶段的地址转换来获得物理地址：**逻辑地址转换**和**线性地址空间分页**。

即使使用最少的段，处理器地址空间中的每个字节都是用一个**逻辑地址**访问的。逻辑地址由一个16位段选择器和一个32位偏移量组成，如下图。段选择器标识字节所在的段，偏移量指定字节在段中相对于段基地址的位置。

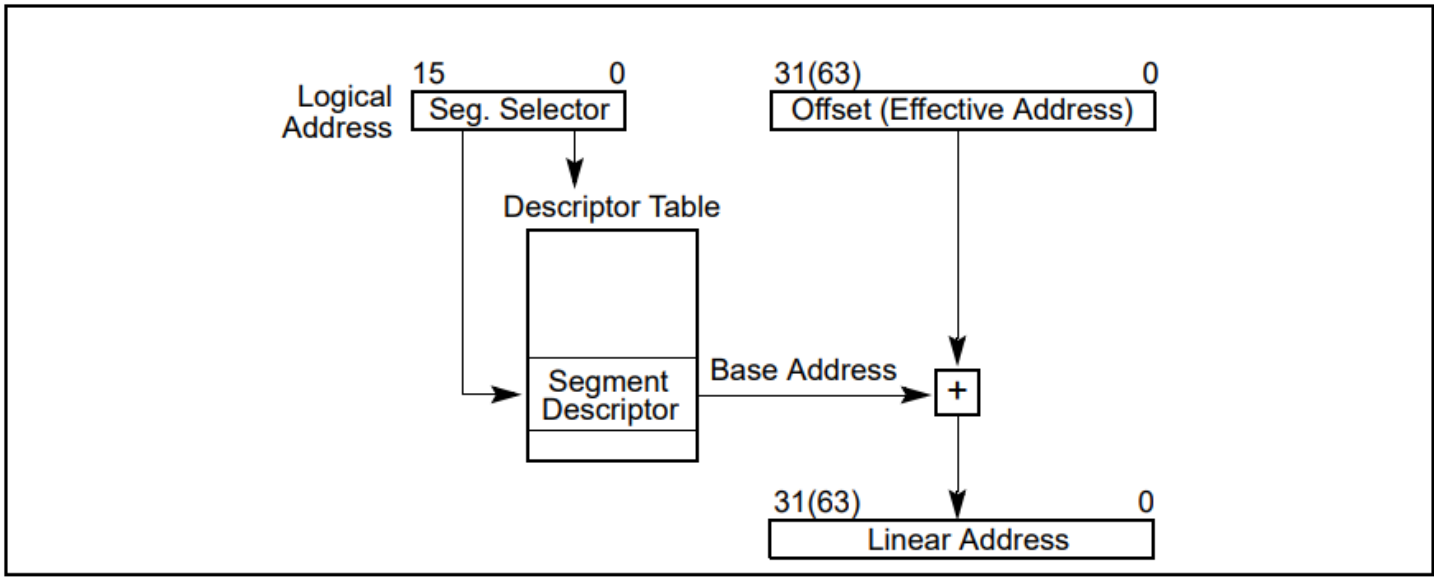

Figure 3-5. Logical Address to Linear Address Translation

处理器将逻辑地址翻译为**线性地址**，就是处理器线性地址空间中的32位地址。与物理地址空间一样，线性地址空间是平坦的（未分段的），2^{32}字节的地址空间，地址范围从0到FFFFFFFFH。线性地址空间包含所有为系统定义的段和系统表。

处理器进行逻辑地址转换的步骤：

1. 使用段选择器中的偏移量来定位GDT或LDT中段的段描述符，并将其读入处理器。（仅当新段选择器被加载到段寄存器中时进行此步。）
2. 检测段描述符获得段的访问权限和范围，确保段是可访问的，并且偏移量在段限制内。
3. 将段描述符中的段基地址与偏移量相加形成线性地址。

如果未启用分页，处理器将线性地址直接映射到物理地址（即线性地址在处理器地址总线上发出）。若线性地址空间被**分页**，则启用**第二级地址转换：线性地址到物理地址**。

4. 描述符的分类

4.1 代码/数据段描述符

若段描述符中的**S标志**为1，则该段为**代码或数据段**，由type字段的最高位（见图3-8，描述符第二个双字的第11位）确定其为数据段（0）还是代码段（1）。

Table 3-1. Code- and Data-Segment Types

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read, conforming
15	1	1	1	1	Code	Execute/Read, conforming, accessed

4.1.1 数据段描述符（Data segment Descriptor）

数据段描述符的type字段的低三位被解读为A（访问过accessed）、W（允许写入write-enable）、E（扩展方向expansion-direction），见上表。

- W位控制数据段是只读段还是读写段。
- A位表示自操作系统或executive上次清除该位后，该段是否被访问过，处理器在将段选择器加载到段寄存器时将其置1，假设包含段描述符的内存类型支持处理器写操作。在显式清除之前，该位保持设置。这个位既可以用于虚拟内存管理，也可以用于调试。

栈段是必可读写的代码段，用不可写的代码段描述符加载段寄存器会引发#GP异常。如果栈段需要动态的大小，其可为可向下扩展数据段（E=1），此时改变栈段大小会在栈底增加栈空间。如果栈段会保持静态大小，那么此栈段既可能是向上扩展段，也可能是向下扩展段。

所有数据段都是**不一致的**，即不允许更低特权等级的程序或过程访问。与代码段不同，数据段不需要特殊的访问门，即可被更高特权的程序或过程访问。

*此处，**更高特权等级的程序或过程**是指**在数字更小的特权等级下执行的代码。***

4.1.2 代码段描述符（Code segment Descriptor）

代码段描述符的type字段的低三位被解读为A、R（可读性read-enable）、C（一致性conforming），见上表。

- 代码段可以是**只可执行的**，或**可执行/可读的**，取决于R位。当常量或其他静态数据与指令代码一起放置在ROM中时，可能会使用执行/读取段。处理器可以通过使用带有CS覆盖前缀的指令或通过在数据段寄存器（DS、ES、FS或GS寄存器）中加载代码段的段选择器，来从代码段读取数据。保护模式下，代码段不可写。
- 代码段既可以是**一致的**（conforming），也可以是**不一致的**（nonconforming）。处理器切换到更高特权等级的一致代码段开始执行的时候，当前特权等级的代码段可以被允许继续执行。处理器切换到不同特权等级的不一致代码段会产生#GP异常，除非使用调用门或者任务门。不访问受保护设备或某些异常处理程序（如除法错误或溢出）的系统应用可能会被加载进一致代码段，而需要保护其不受特权等级较低的程序和过程影响的系统应用应放置在不一致的代码段中。
- 不能通过调用或者跳转的方式从当前特权等级的代码段切换到更低等级（数字更大）的代码段上执行，无论目标段一致性如何，否则会产生#GP异常。

如果GDT或LDT中的段描述符位于ROM且软件或处理器试图更新（写入）这些段描述符，处理器可能会进入无限循环。为防止此状况，需要在ROM中的所有段描述符置A位为1，并删除试图修改位于ROM中的段描述符的操作系统或executive的代码。

4.2 系统段描述符

若段描述符S位为0，则为**系统描述符**，分为**两类**：

系统段描述符，指向系统段：LDT、TSS：

- 局部描述符表描述符 Local descriptor-table (LDT) segment descriptor
- 任务状态段描述符 Task-state segment (TSS) descriptor

门描述符，即**门**，保存了指向代码段过程入口点的指针（调用门、中断门和陷阱门），或者持有TSS的段选择器（任务门）：

- 调用门描述符 Call-gate descriptor
- 中断门描述符 Interrupt-gate descriptor
- 陷阱门描述符 Trap-gate descriptor
- 任务门描述符 Task-gate descriptor

Table 3-2. System-Segment and Gate-Descriptor Types

Type Field					Description	
Decimal	11	10	9	8	32-Bit Mode	IA-32e Mode
0	0	0	0	0	Reserved	Upper 8 byte of an 16-byte descriptor
1	0	0	0	1	16-bit TSS (Available)	Reserved
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-bit TSS (Busy)	Reserved
4	0	1	0	0	16-bit Call Gate	Reserved
5	0	1	0	1	Task Gate	Reserved
6	0	1	1	0	16-bit Interrupt Gate	Reserved
7	0	1	1	1	16-bit Trap Gate	Reserved
8	1	0	0	0	Reserved	Reserved
9	1	0	0	1	32-bit TSS (Available)	64-bit TSS (Available)
10	1	0	1	0	Reserved	Reserved
11	1	0	1	1	32-bit TSS (Busy)	64-bit TSS (Busy)
12	1	1	0	0	32-bit Call Gate	64-bit Call Gate
13	1	1	0	1	Reserved	Reserved
14	1	1	1	0	32-bit Interrupt Gate	64-bit Interrupt Gate
15	1	1	1	1	32-bit Trap Gate	64-bit Trap Gate

系统描述符的type字段对应的描述符见上表。有关门描述符和TSS描述符的内容会在之后的读书笔记学习，此处，为学习LDT描述符，按照手册的内容，接下来是一些对系统描述符表的介绍。

4.2.1 系统描述符表

描述符表是**段描述符的数组**，其长度多变，最多可以装有8192（2^{13}）个8字节的段描述符，分为两种：GDT和LDT。

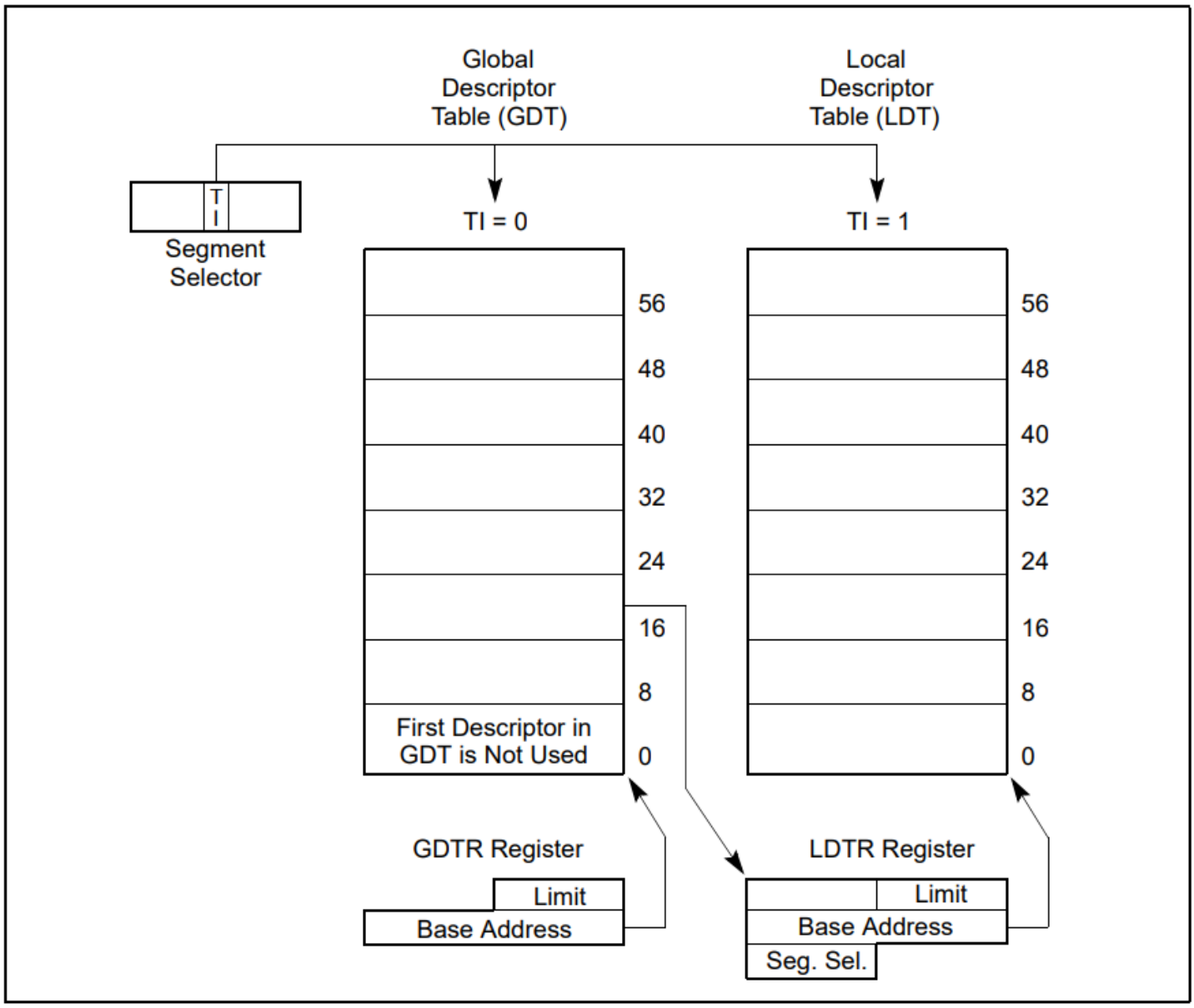

Figure 3-10. Global and Local Descriptor Tables

每个系统都必须定义一个GDT，为系统中所有程序和任务所使用。可选地，可以定义一个或多个LDT，如为每个运行的任务定义一个LDT，或，几个或者所有的任务共享LDT。

GDT本身不是段，而是线性地址空间中的一个数据结构，其线性基地址和限制必须载入GDTR寄存器，GDT的基地址应在8字节边界上对齐，以获得最佳处理器性能。GDT的表限制值使用字节为单位表示。与段一样，处理器可以将表限制值与GDT基地址相加，以获得GDT最后一个有效字节的地址。如果将GDT表限制设为0，则会导致GDT只有一个有效字节。因为段描述符总是8个字节长，所以GDT的表限制应该总是比8的整数倍小1（即8N-1）。

处理器不使用GDT中的第一个描述符。此“空描述符”的段选择器在被加载到数据段寄存器（DS、ES、FS 或 GS）时不会生成异常，但在尝试使用该描述符访问内存时，将会生成一般保护异常（#GP）。通过使用该段选择器初始化段寄存器，可以确保在意外引用未使用的段寄存器时产生异常报告。

LDT位于LDT类型的系统段中。GDT必须包含LDT段的段描述符。如果系统支持多个LDT，则每个LDT都必须有一个单独的段选择子和GDT中的段描述符。

LDT需要使用相应的段选择器进行访问。为了在访问LDT时消除地址转换，LDT的段选择子、线性基地址、限制和访问权限需要存储在 LDTR 寄存器中。

当GDTR寄存器中的数据被存储时（SGDT指令），一个48位的“伪描述符”将被存储在内存中（如下图所示）。为了避免用户模式（特权等级3）中的对齐检查错误，伪描述符应位于奇字地址（即地址值模4为2）。这使得处理器存储的是一个对齐的字，随之是一个对齐的双字。用户态程序通常不存储伪描述符，但是通过这种方式对齐伪描述符可以避免产生对齐检查错误。使用SIDT指令存储IDTR寄存器时，应使用相同的对齐方式。当存储LDTR或任务寄存器（分别使用 SLDT 或 STR 指令）时，伪描述符应位于双字地址（即地址模4为0）。

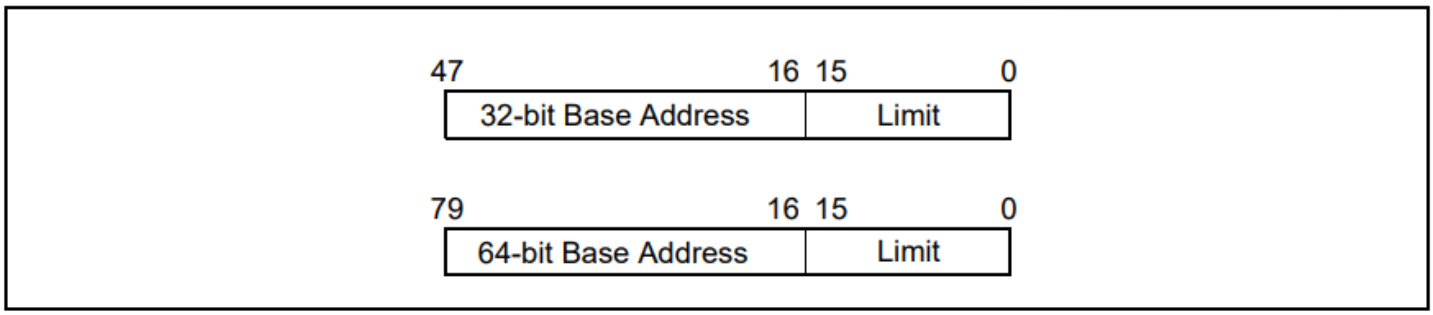

Figure 3-11. Pseudo-Descriptor Formats