HIT-OS读书笔记（二）：保护模式内存管理

1.内存管理概览

IA-32架构的内存管理分为两部分：**分段**和**分页**。

> *在保护模式下,分段机制是必须的，分页机制则是可选的。*
> *而当分页在操作系统或执行程序中正确实现时，物理内存和磁盘之间的页面交换对于程序的正确执行是**透明**的。*

- **分段机制**为每个程序或者任务提供**单独**的代码、数据和栈模块，也可用于保存系统数据结构(如TSS或LDT)，以便多个程序(或任务)可以在同一处理器上运行而不会相互干扰。
- **分页机制**实现了传统的请求调页**虚拟内存**系统，在这种系统中，程序执行环境中的各个部分根据需要映射到物理内存中，分页还可用于在**多个任务之间**提供隔离。

这两种机制(分段和分页)可以配置为支持简单的单程序(或单任务)系统、**多任务系统或**使用**共享内存的多处理器系统**。

分段机制将线性地址空间划分为更小的**受保护**地址空间。在多任务系统或共享内存的多处理器系统，每个程序/任务可以分配私有的一组段。处理器强制规定段之间的边界，确保一个程序不会通过写入另一个程序的段来干扰另一个程序的执行。同时，分段机制还对**段的类型进行了区分**，以便可以**限定**在特定类型的段上执行的**操作**。

多任务计算系统通常定义的线性地址空间比一次性将所有地址都包含在物理内存中的经济可行性大得多，因此需要某种“虚拟化”线性地址空间的方法。分页支持一种“虚拟内存”环境，在这种环境中，使用**少量物理内存(RAM和ROM)和一些磁盘存储来模拟大型线性地址空间**。

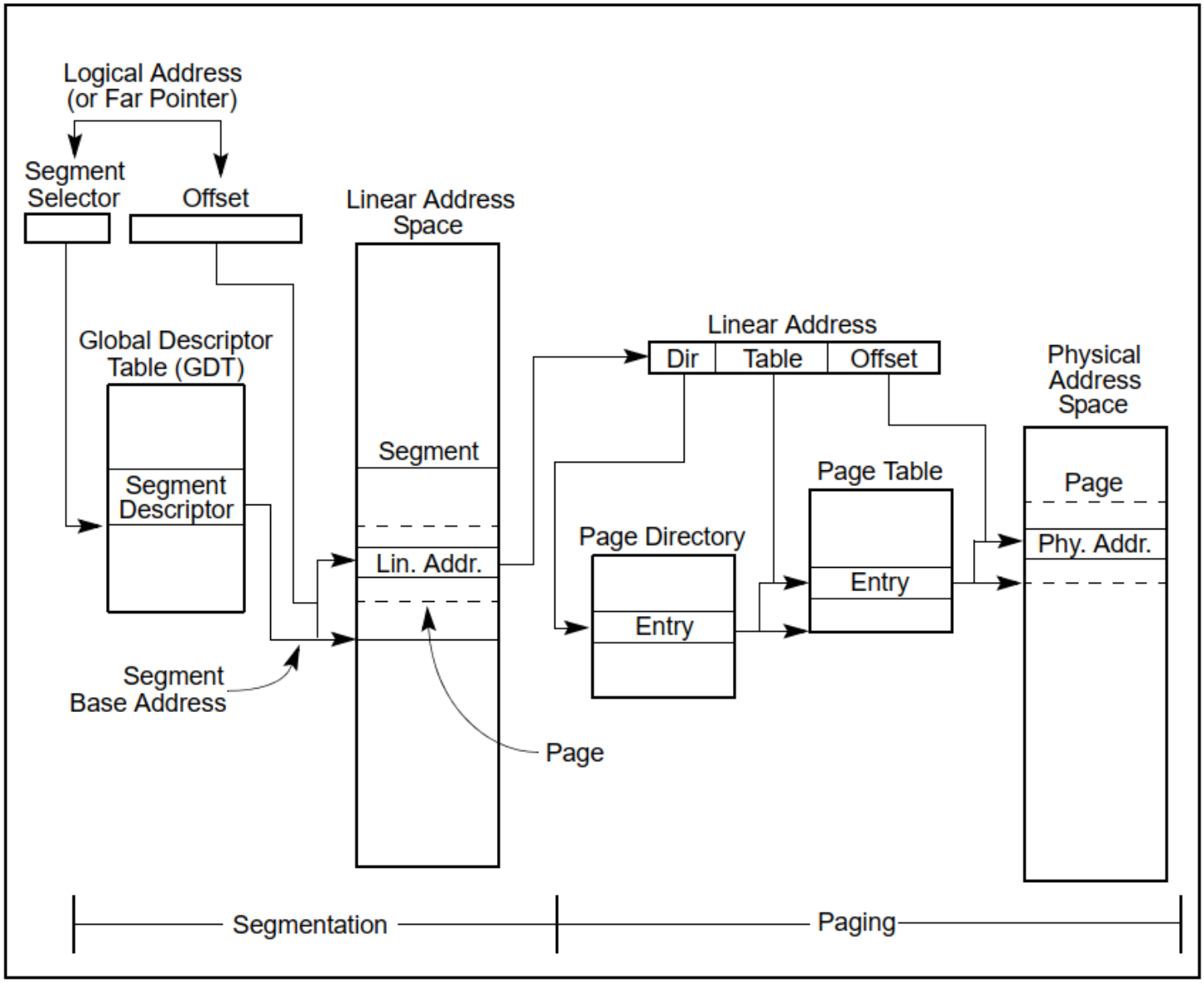


Figure 3-1. Segmentation and Paging

各类地址区分及其变换

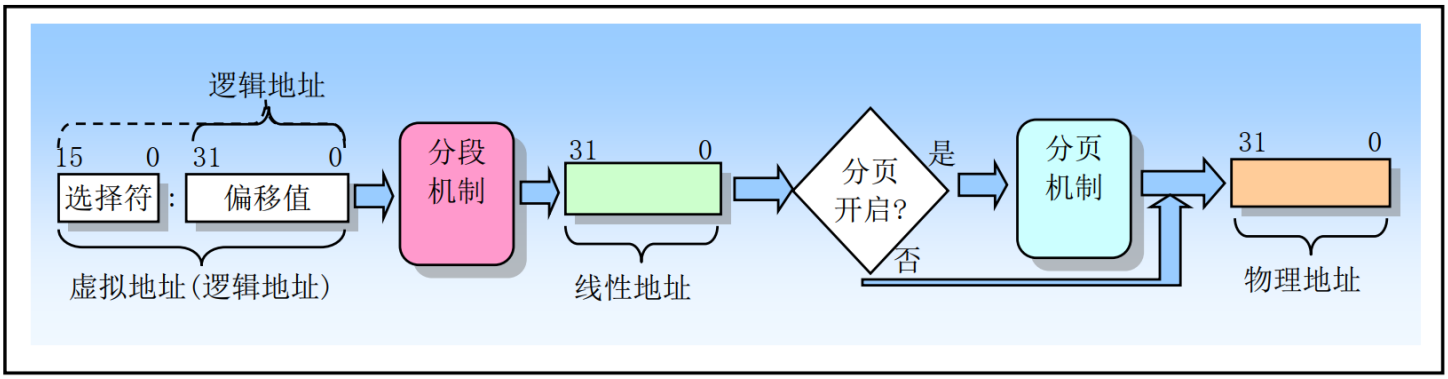
地址	解释
逻辑地址	进程代码访问内存的地址空间，是由程序生成的地址， 由16位段选择子和32位偏移量组成，**与实际物理内存无关**， 是在程序内部使用的抽象地址
线性地址	**处理器可寻址内存空间**，逻辑地址经过分段、分页机制转换后得到的地址， 由描述符和偏移量组成，地址空间中整数连续，是虚拟地址空间的地址， 地址空间中整数连续，但不一定映射到物理内存的连续位置上。
虚拟地址	应用程序对内存的地址请求，即程序所看到的地址， 也就是逻辑地址和线性地址的总和

地址	解释
物理地址	**实际存在**于内存中的地址，通过页表机制由线性地址转换而来。 用于**内存芯片级**的单元寻址，与处理器和CPU连接的地址总线相对应， 可以映射到读写内存、只读内存和内存映射I/O。

注：

- 系统中的**所有段**都包含在处理器的线性地址空间中，要定位特定段中的字节，**必须提供逻辑地址**。
- 线性地址是处理器线性地址空间中的32位地址，定义一个未分段的4GB的地址空间，地址范围为 `0~FFFFFFFFH` ，地址空间中包含**所有为系统定义的地址段和系统表**。
- 在保护模式下，IA-32架构提供的正常物理地址空间为未分段的4GB，地址范围为 `0~FFFFFFFFH` 。从Pentium Pro处理器开始，IA-32架构还支持将物理地址空间扩展到64GB,此扩展可通过以下两种方式调用:
  - 使用物理地址扩展(PAE)标志，位于控制寄存器CR4的第5位。
  - 使用36位页面大小扩展(PSE-36)功能(在Pentium III处理器中引入)。

在保护模式下的系统架构中，处理器通过两个阶段的地址转换得到物理地址：**逻辑地址转换**和**线性地址空间分页**。



> **逻辑地址到线性地址的变换**
>
> - 完整的逻辑地址包含段选择符和段内偏移地址两部分。
> - 段选择符选择对应的段，将段首地址作为基地址加上偏移地址即可将逻辑地址映射到线性地址空间。

> **线性地址到物理地址的变换**
>
> - 如果不使用分页机制，处理器的线性地址空间**直接映射**到处理器的物理地址空间。
> - 如果使用分页机制
>   - 在页式存储管理方式中地址结构由两部构成，前一部分是虚拟页号(VPN)，后一部分为虚拟页偏移量(VPO)。

    - 页号用于在页表中查找对应的页表项，页表项中包含了该虚拟页所映射的物理页号以及访问权限等信息，通过将物理页号和页内偏移量组合得到最终的物理地址。
- 如果被访问的页面当前不在物理内存中，处理器将中断程序的执行(通过生成页面错误异常)。然后，操作系统或执行程序将该页从磁盘读入物理内存，并继续执行程序。

分段和分页是两种不同的地址变换机制，它们都对整个地址变换操作提供独立的处理阶段。尽管两种机制都使用存储在内存中的变换表，但所用的表结构不同。实际上，段表存储在线性地址空间，而页表则保存在物理地址空间。

2.分段机制

IA-32架构支持的分段机制可用于实现各种各样的系统设计。这些设计的范围从平滑模型(只使用最小的分段来保护程序)到多分段模型(使用分段来创建一个健壮的操作环境，使多个程序和任务可以可靠地执行)。

2.1 Basic Flat Model基础平滑模型

系统最简单的内存模型是基本的“平滑模型”，在这种模型中，操作系统和应用程序可以访问**连续的、未分段的**地址空间。

> *基本平面模型在最大程度上对系统设计人员和应用程序编程人员隐藏了系统架构的分段机制。*

在IA-32架构中实现基本的平面内存模型，至少必须创建两个段描述符。

- 一个用于引用代码段
- 一个用于引用数据段

但是这两个段**都映射到整个**线性地址空间，即两个段描述符具有相同的基址值0和相同的段限制4GB。通过将段限制设置为4GB，即使没有物理内存驻留在特定地址上，分段机制也不会为超出限制的内存引用生成异常。在此模型中，ROM和RAM的位置也相对固定。

- ROM (EPROM)通常位于物理地址空间的顶部，因为处理器在 `FFFF FFF0H` 处开始执行。
- RAM (DRAM)被放置在地址空间的底部，因为重置初始化后DS段的初始基址是0。

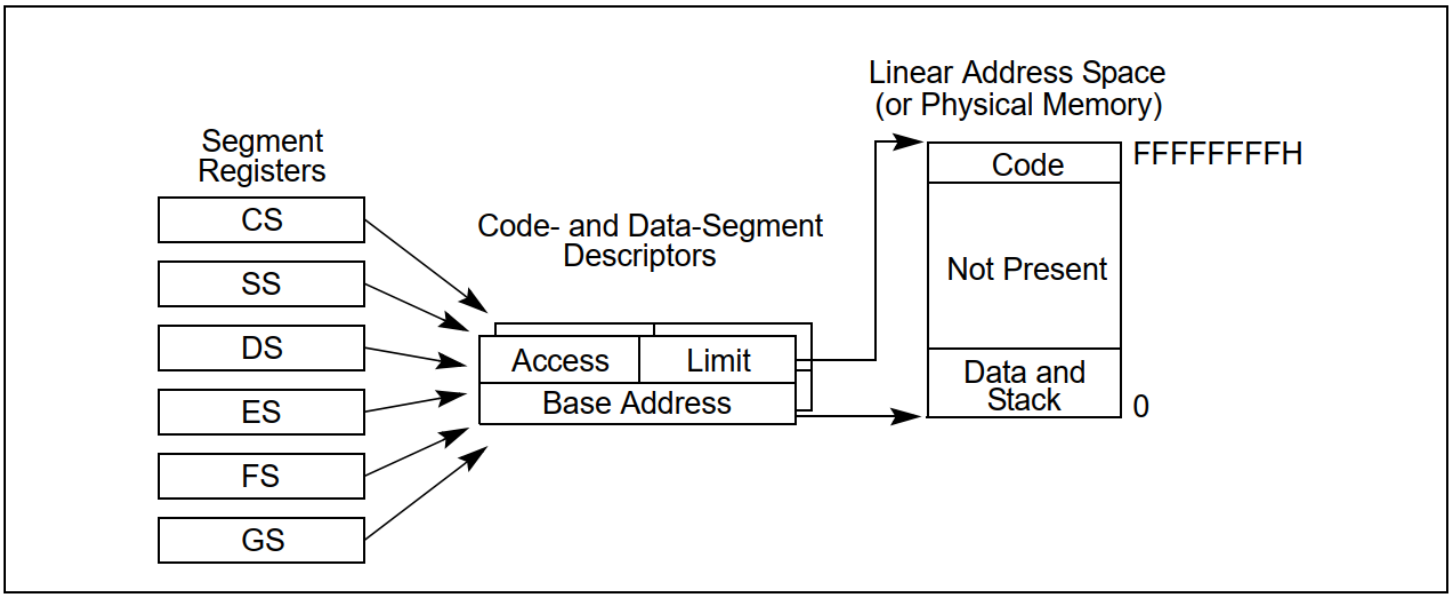


Figure 3-2. Flat Model

2.2 Protected Flat Model受保护的平滑模型

受保护的平滑模型能够提供了最低级别的**硬件保护**，以防止某些类型的程序错误。其与基本平滑模型类似，而又提出了改进。

- 将段界限仅设置为**实际物理内存地址范围内**
- 在任何访问不存在的内存的尝试时生成一个通用保护异常(#GP)。

同时，还可以向这个受保护的平面模型添加更多的复杂性，以提供更多的保护。

为了让分页机制分离用户和内核程序代码和数据，可以定义相互覆盖且从线性地址空间中的地址0开始的四个段：

- 权限级别3的用户代码和数据段
- 权限级别0的内核代码和数据段。

这种平滑分段模型以及简单的分页设置可以保护**操作系统不受应用程序**的影响，同时通过为每个任务或进程添加**单独的**分页结构，还可以保护**应用程序不受其他应用程序**的影响

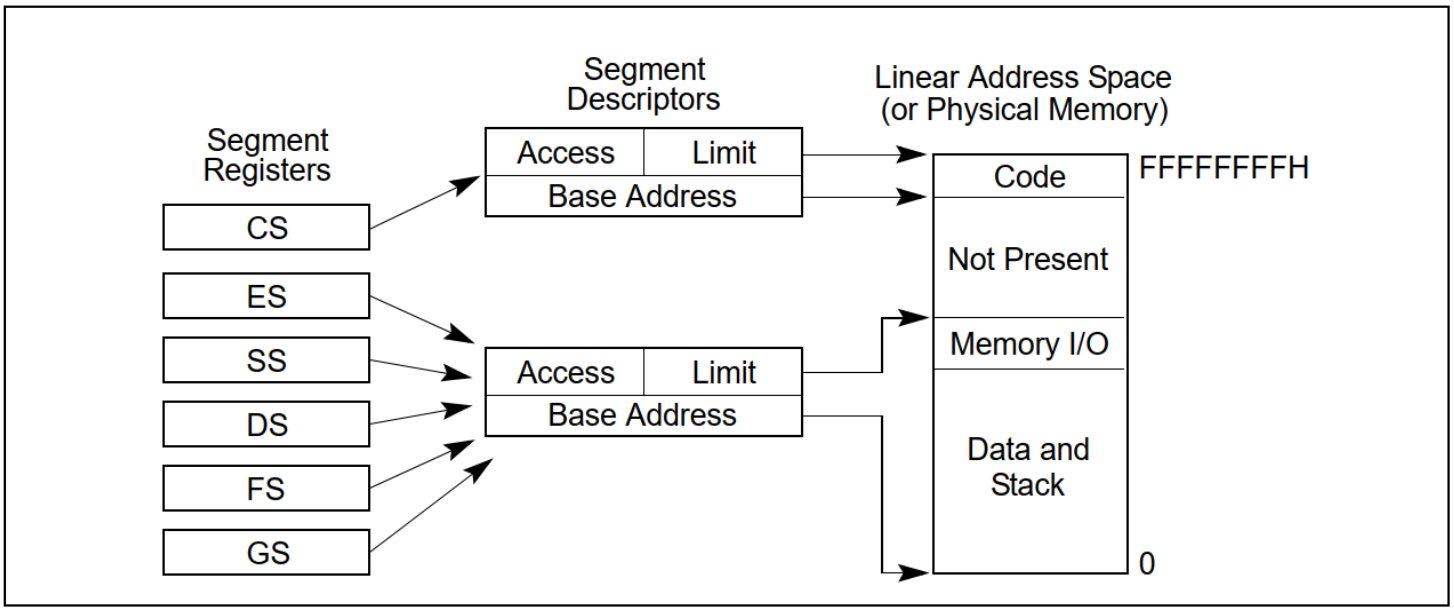


Figure 3-3. Protected Flat Model

2.3 Multi-Segment Model多分段模型

多分段模型充分利用了分段机制的全部功能，为**代码、数据结构、进程和任务**提供硬件强制保护。

多分段模型中，每个进程(或任务)都有自己的段描述符表和自己的段。**进程可以完全私有这分配的段，也可以与其他进程间进行共享。**

> *对系统上运行的各个程序的所有段和执行环境的访问由硬件控制。*

多分段模型的**访问检查**机制能够

- 防止引用段界限之外的地址
- 防止在某些段中执行不允许的操作。例如，对于只读代码段，可以使用硬件来防止对代码段的写入。

为段创建的访问权限信息还可以用于设置保护环或保护级。保护级可用于保护操作系统进程免受应用程序**未经授权的访问**。

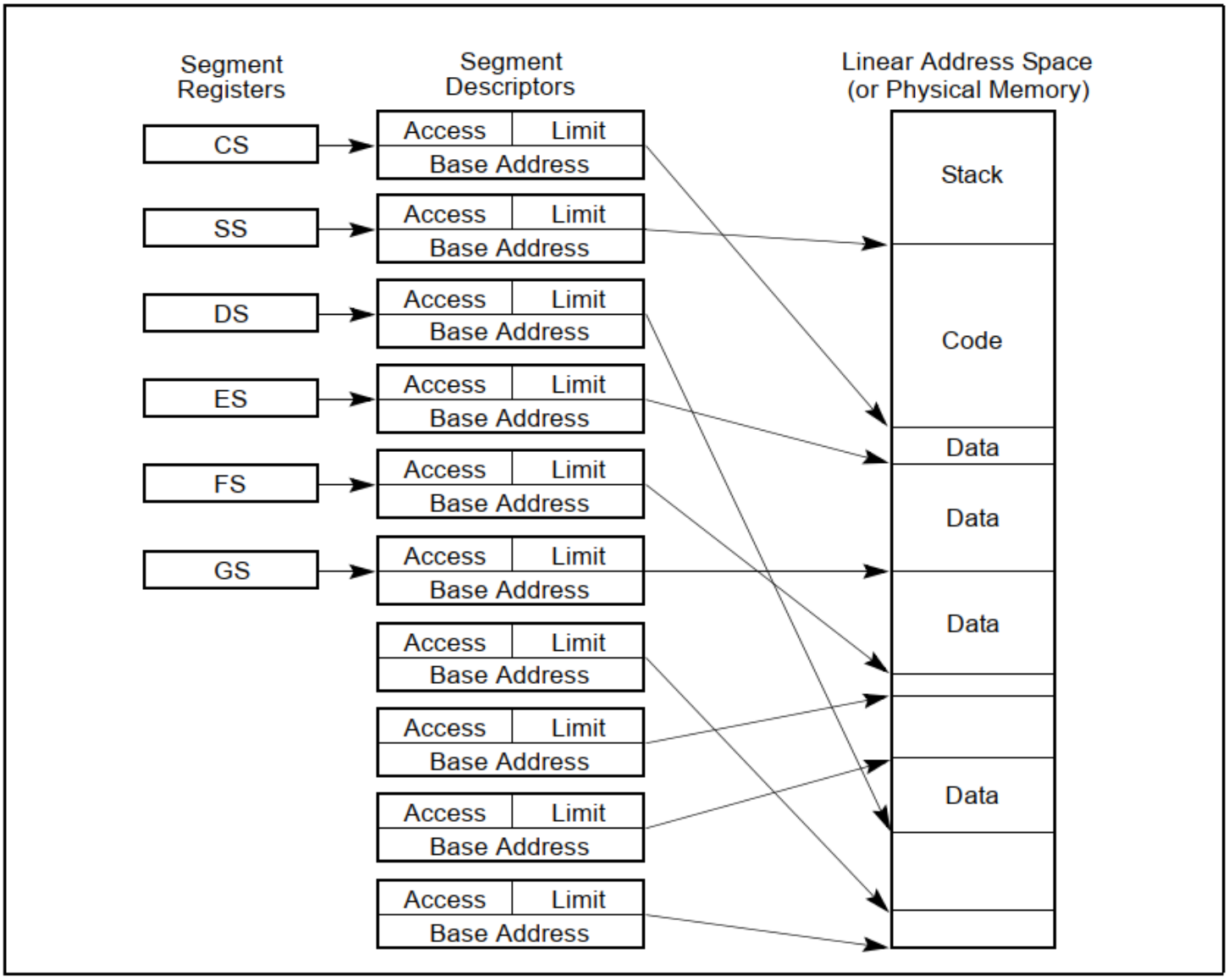


Figure 3-4. Multi-Segment Model

3.逻辑地址和线性地址的转换

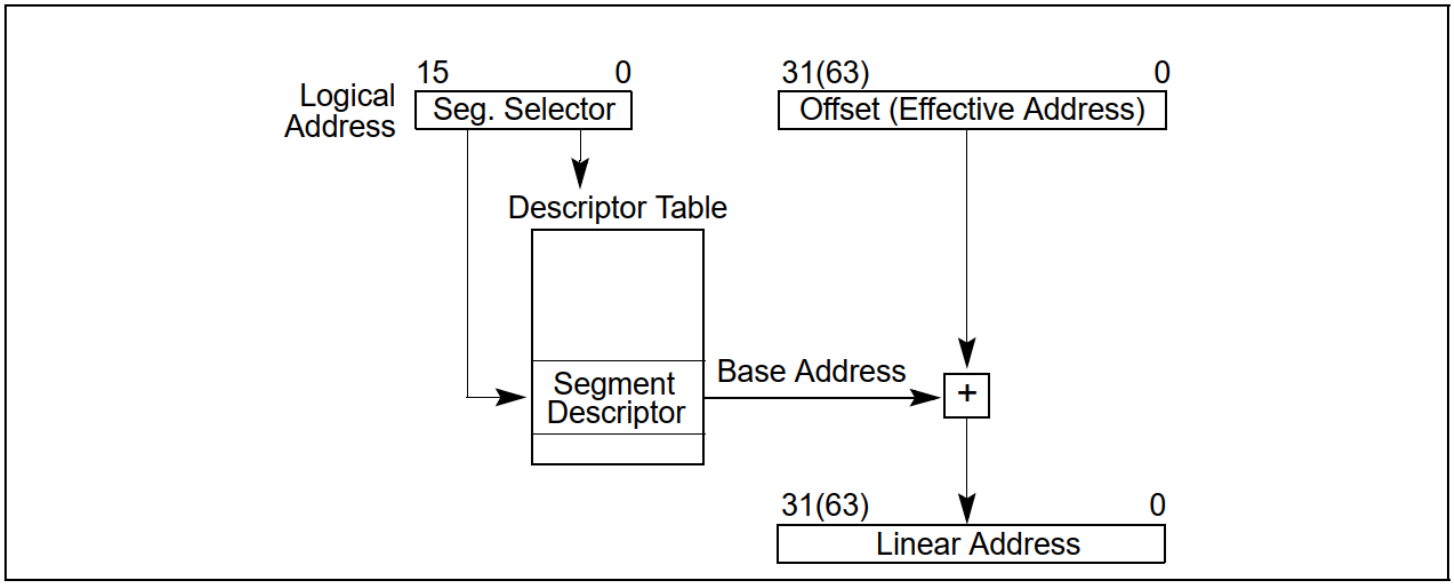


Figure 3-5. Logical Address to Linear Address Translation

为了把逻辑地址转换成一个线性地址，处理器会执行以下操作：

> 1. 使用段选择符中的偏移值（段索引）在GDT或LDT表中**定位**相应的段描述符。（仅当一个新的段选择符加载到段寄存器中时才需要这一步。）
> 2. 利用段描述符检验段的访问权限和范围，以确保该段是**可访问的**并且偏移量位于段**界限内**。
> 3. 把段描述符中取得的**段基地址加到偏移量**上，最后形成一个线性地址。

3.1 Segment Selectors段选择子

> *段选择子作为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赋值或修改，而不是应用程序。*

段选择子是一个16位的段标识符，**不直接指向段**而指向定义段的段描述符。一个段选择子包含以下部分:

- **索引Index**：对应GDT或LDT中8192个描述符中的其中一个。计算公式为：$Index \times 8 + Address_{GDT/LDT}^{Base}$
- **表选择符TI**：指定使用的描述符表，TI为0选择GDT，TI为1选择当前LDT。
- **请求权限级别RPL**：指定选择器的特权级别。特权级别的取值范围为0~3，其中0为最高特权级别。

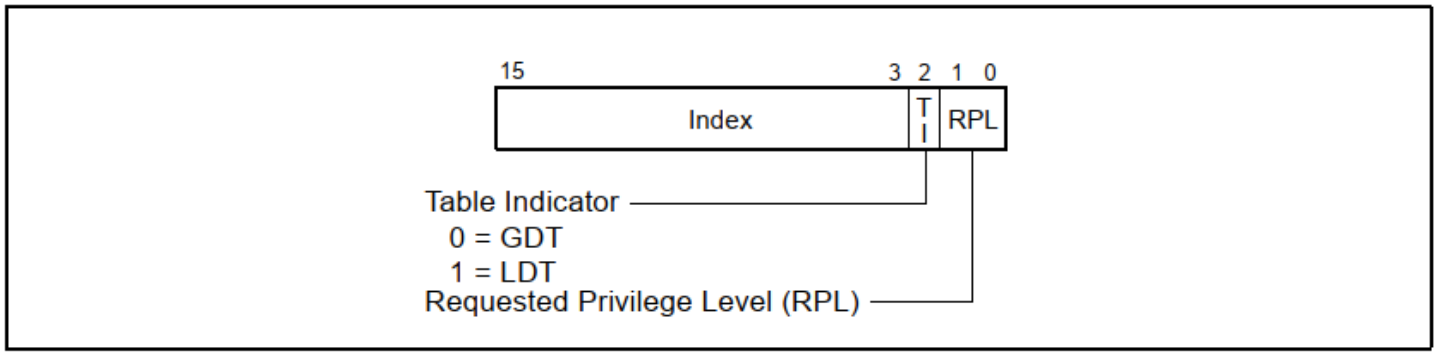


Figure 3-6. Segment Selector

处理器不使用GDT的第一个条目。指向GDT的此条目的段选择子用作“空段选择子”。

- 当段寄存器(除了CS或SS寄存器)被加载为“空段选择子”时，处理器不会产生异常，因而可用于初始化未使用的段寄存器。
- 但是，当使用持有“空段选择子”的段寄存器访问内存时会产生异常。用空段选择子加载CS或SS寄存器也会导致生成一个通用保护异常(#GP)。

3.2 Segment Registers段寄存器

为了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。段寄存器中的**每一项都支持特定类型的内存引用**(代码、堆栈或数据)。

实际上，对于任何类型的程序执行，至少**代码段(CS)、数据段(DS)和堆栈段(SS)寄存器**必须装入有效的段选择子。处理器还提供了三个额外的数据段寄存器(ES、FS和GS)，它们可以用来为当前正在执行的程序(或任务)提供额外的数据段。

> *对于一个程序访问一个段，段的段选择子必须已经加载到一个段寄存器中。因此，尽管系统可以定义数千个段，但**只有6个可以立即使用**。其他段可以通过在程序执行期间将它们的段选择子加载到这些寄存器中来使用。*

每个段寄存器都有一个“可见”部分和一个“隐藏”部分。当段选择器被加载到段寄存器的可见部分时，处理器也将段选择器指向的段描述符的基址、段界限和访问控制信息加载到段寄存器的隐藏部分。缓存在段寄存器中的信息**均允许**处理器进行转换地址，而**不需要额外的总线周期**从段描述符读取基址和界限。

> *在多个处理器访问同一个描述符表的系统中，当描述符表被修改时，软件需要重新加载段寄存器。否则，缓存在段寄存器中的旧段描述符可能会在其原始内存版本被修改后被使用。*

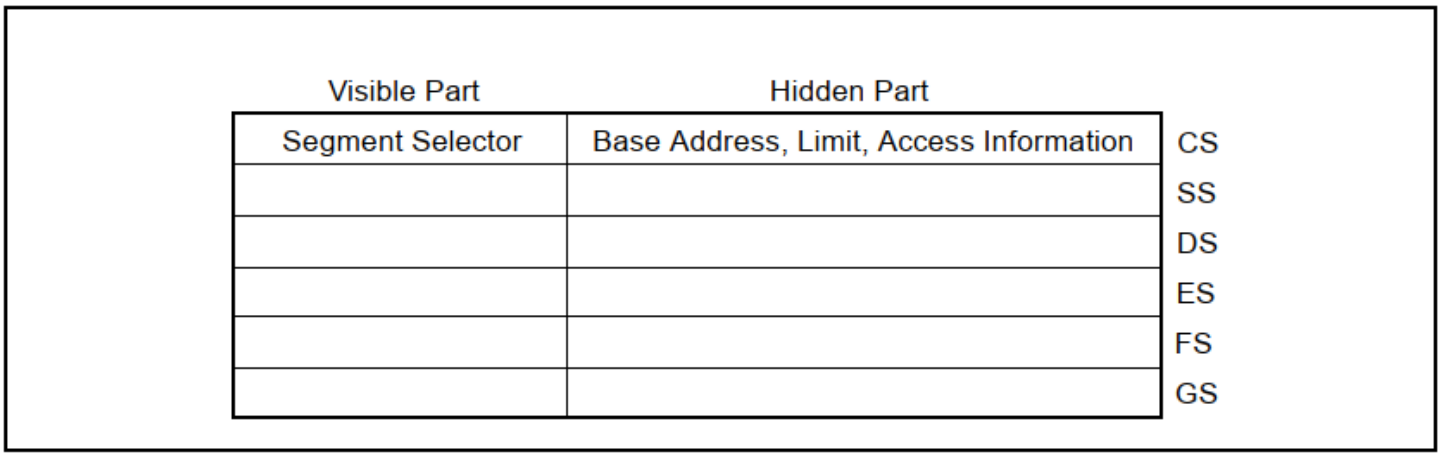


Figure 3-7. Segment Registers

系统提供了两种加载指令来加载段寄存器:

- 直接加载指令：如 `MOV` 、 `POP` 、 `LDS` 、 `LES` 、 `LSS` 、 `LGS` 和 `LFS` 指令。这些指令**显式地引用段寄存器**。
- 隐式加载指令，例如 `CALL` 、 `JMP` 和 `RET` 指令的跳转指针版本， `SYSENTER` 和 `SYSEXIT` 指令，以及 `IRET` 、 `INT n` 、 `INTO` 和 `INT3` 指令。这些指令**改变CS寄存器**(有时是其他段寄存器)的内容，作为其操作的附带部分。
- `MOV` 指令也可用于在通用寄存器中存储段寄存器的可见部分。

3.3 Segment Descriptors段描述符

段描述符是GDT或LDT中的一种数据结构，它向处理器提供**段的大小和位置，以及访问控制和状态信息**。段描述符通常由编译器、链接器、加载器、操作系统或执行程序创建，而**不是由应用程序创建**。

段描述符中的**标志和字段**如下:

- **段界限Segment Limit**：指定**段的大小**，处理器把两个段限制字段放在一起形成一个20位的值，与G有关联。
- **段基地址Base Addrress**：在4GB的线性地址空间中定义段的**字节0的位置**。处理器将三个基址字段放在一起形成一个32位值。段基址应该对齐到16字节的边界，从而允许程序通过在16字节边界上对齐代码和数据来最大化性能。
- **类型字段Type**：指示**段或门的类型**，并指定可以对该段进行的**访问类型和增长方向**。此字段的解释取决于描述符类型标志是指定应用程序(代码或数据)描述符还是系统描述符。
- **描述符类型S**：指定段描述符是用于**系统段**(S为0)还是用于**代码段或数据段**(S为1)。
- **描述符特权级别DPL**：指定段的**权限级别**，用于控制对段的访问。特权级别可以从0到3，其中0是最高级别的特权级别。
- **段存在标识P**：指示段是否存在于内存中(P为1)或不存在(P为0)。当P为0时，操作系统或执行器可以自由地使用标记为“Available”的位置来存储自己的数据，例如关于丢失段的位置的信息。

    - 如果P为0，当一个指向段描述符的段选择器被加载到段寄存器中时，处理器会产生一个**段不存在异常**(#NP)。
    - 内存管理软件可以使用这个标志来控制哪些段在给定时间实际加载到物理内存中。可用于管理虚拟内存。
- **默认操作大小/默认堆栈指针大小和/或上限标志D/B**：根据段描述符是可执行代码段、向下展开数据段还是堆栈段，执行不同的功能。
    - **可执行代码段**：段中指令引用的有效地址和操作数的**默认长度。**
        - 如果为1，则采用32位地址和32位或8位操作数;
        - 如果为0，则假定16位地址和16/8位操作数。

        > *指令前缀66H可用于选择非默认的操作数长度，前缀67H可用于选择非默认的地址长度。*

    - **堆栈段(由SS寄存器指向的数据段)**：指定用于隐式堆栈操作(如 `push` 、 `pop` 和 `call` 等)的堆栈**指针的大小。**
        - 如果为1，则使用32位堆栈指针，该指针存储在32位ESP寄存器中;
        - 如果为0，则使用16位堆栈指针，该指针存储在16位SP寄存器中。
    - **向下展开数据段**：指定了**段的上界。**
        - 如果为1，则上限为 `FFFFFFFFH` ；
        - 如果为0，则上限为 `FFFFH` 。
- **粒度位G**：决定段界限**单位及扩展范围**
    - G为0，段界限以字节为单位，段的扩展范围1B~1MB，以字节为单位递增
    - G为1，段界限以4KB为单位，段的扩展范围4KB~4GB，以4kbyte为单位递增
- **64位代码段L**：表示代码段是否包含本机64位代码。值为1表示该代码段中的指令以64位模式执行，值为0表示该代码段中的指令以兼容模式执行。
    - 如果L位为1，则D位必须为0。当不处于IA-32e模式或非代码段时，第21位被保留并应始终设置为0。
- **可用和保留比特位AVL**：一般用于操作系统或应用程序自定义的目的。

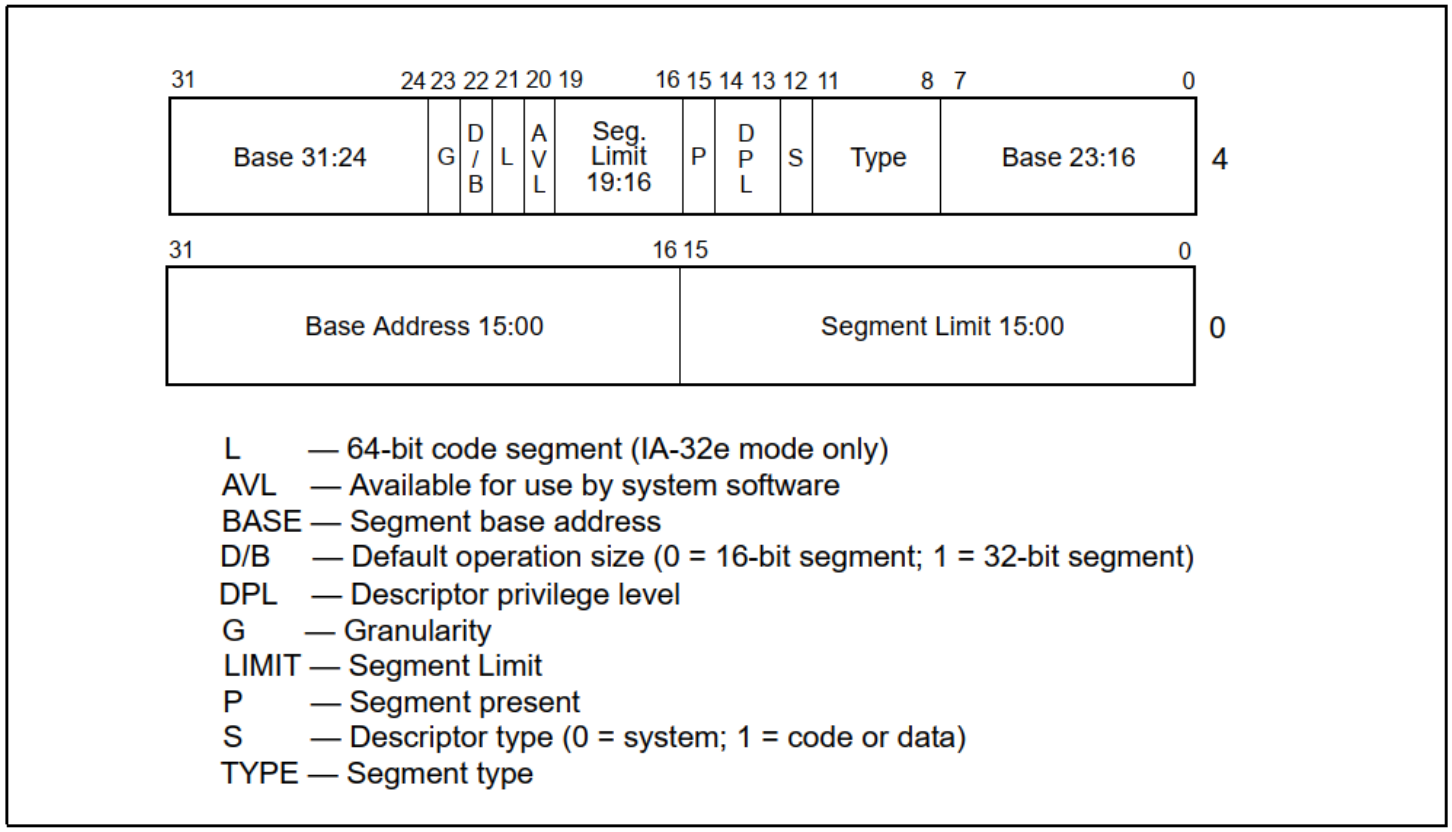
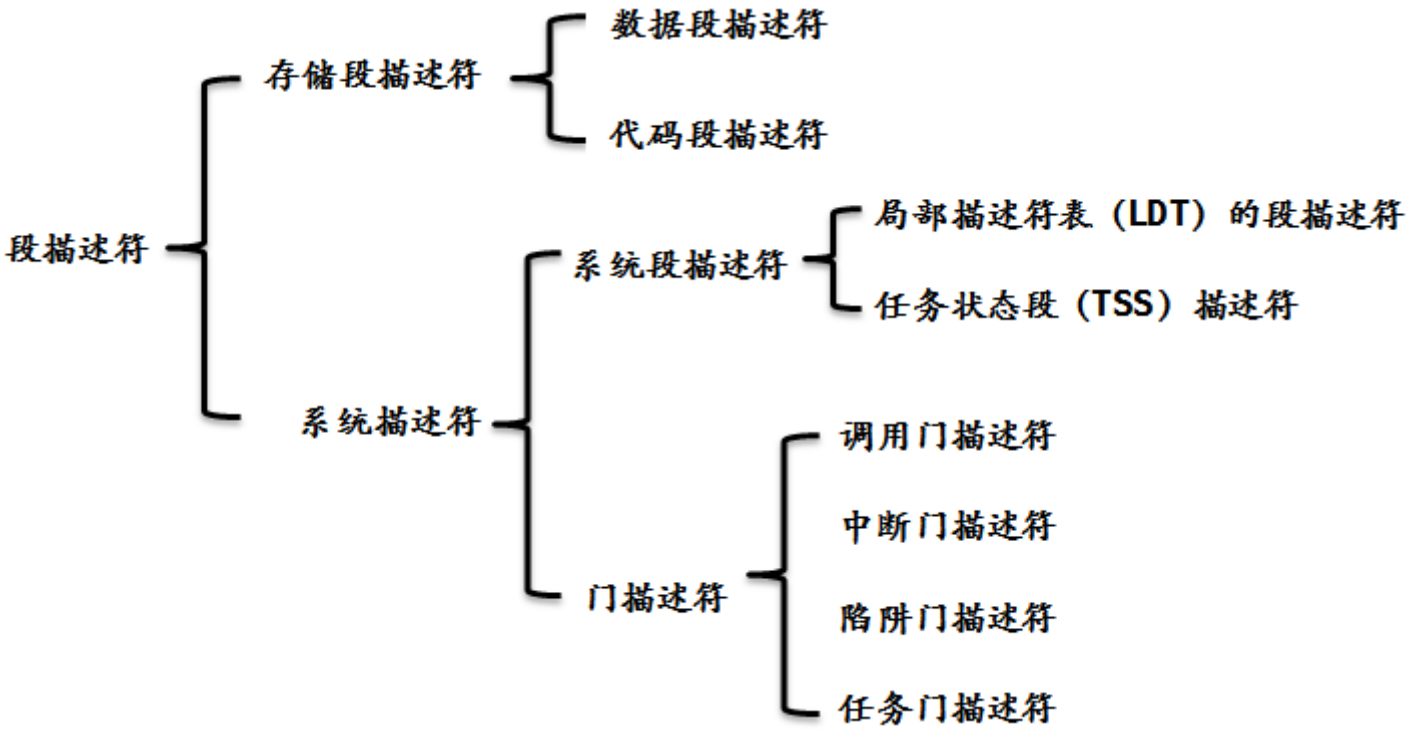


Figure 3-8. Segment Descriptor

4.描述符的分类



4.1 代码和数据段描述符类型

4.1.1 数据段描述符

当段描述符中的S(描述符类型)标志为1且TYPE字段的最高位为0时，表明是一个数据段描述符。

- 此时D/B位取B，指明隐含堆栈操作时的栈指针大。
- Type字段从高到低依次为 `0 E M A`，低3位被分别用于表示已访问A（Accessed）、可写W（Write-enable）和扩展方向E（Expansion-direction）
  - A位用于指明从上次操作系统清零该位后一个段是否被访问过。
  - W位指示段的读写属性。
    - W为0：段不允许写入，否则会引发处理器异常中断；
    - W为1：允许写入。
  - E位指示段的扩展方向
    - E为0：表示向上扩展的段，逻辑地址中的偏移值范围可以从0到Limit；
    - E为1：表示向下扩展的段，逻辑地址中的偏移范围可以从Limit到 `FFFFH` （当B=0时）或者 `FFFF FFFFH` （当B=1时）。

> *堆栈段必须是可读/写的数据段。若使用不可写数据段的选择符加载到SS寄存器中，将导致一个一般保护异常。如果堆栈段的长度需要动态地改变，那么堆栈段可以是一个向下扩展的数据段（扩展方向标志置位）。这里，动态改变段限长将导致栈空间被添加到栈底部。*

4.1.2 代码段描述符

当段描述符中的S(描述符类型)标志为1且TYPE字段的最高位为1时，表明是一个代码段描述符。

- 此时D/B位取D，用于指出该段中的指令引用有效地址和操作数的默认长度。
- Type字段从高到低依次为 `1 C R A`，低3位被解释成已访问 A（Accessed）、可读R（Read-enable）和一致的C（Conforming）
  - A位用于指明从上次操作系统清零该位后一个段是否被访问过.
  - R位指示段的读属性。
    - R为0：代码段不可读，只能执行，用以限制程序的行为；
    - R为1：代码段可读，可执行。
  - C位指示代码段的一致性
    - C为0：表示非一致性代码段。这样的代码段可以被同级代码段调用，或者通过门调用；
    - C为1：表示一致性代码段。可以从低特权级的程序转移到该段执行（但是低特权级的程序仍然保持自身的特权级）。

> *所有的数据段都是非一致性的，即意味着它们不能被低特权级的程序或过程访问。然而与代码段不同，数据段可以被更高特权级的程序或过程访问，而无需使用特殊的访问门。*

表 4-3 代码段和数据段描述符类型

类型（TYPE）字段					描述符类型	说明
十进制	位 11	位 10	位 9	位 8		
		E	W	A		
0	0	0	0	0	数据	只读
1	0	0	0	1	数据	只读，已访问
2	0	0	1	0	数据	可读/写
3	0	0	1	1	数据	可读/写，已访问
4	0	1	0	0	数据	向下扩展，只读
5	0	1	0	1	数据	向下扩展，只读，已访问
6	0	1	1	0	数据	向下扩展，可读/写
7	0	1	1	1	数据	向下扩展，可读/写，已访问
		C	R	A		
8	1	0	0	0	代码	仅执行
9	1	0	0	1	代码	仅执行，已访问
10	1	0	1	0	代码	执行/可读
11	1	0	1	1	代码	执行/可读，已访问
12	1	1	0	0	代码	一致性段，仅执行
13	1	1	0	1	代码	一致性段，仅执行，已访问
14	1	1	1	0	代码	一致性段，执行/可读
15	1	1	1	1	代码	一致性段，执行/可读，已访问

4.2 系统描述符类型

当段描述符中的S(描述符类型)标志为0时，该描述符类型为系统描述符。 处理器识别以下类型的系统描述符:

- 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
- 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

这些描述符类型分为两类:系统段描述符和门描述符。系统段描述符指向系统段(LDT和TSS段)。门描述符本身就是“门”，它存有代码段(调用门、中断门和陷阱门)中过程入口点的指针，或者持有TSS(任务门)的段选择子。

表 4-4 系统段和门描述符类型

类型（TYPE）字段					说明	
十进制	位 11	位 10	位 9	位 8		
0	0	0	0	0	Reserved	保留
1	0	0	0	1	16-Bit TSS (Available)	16 位 TSS（可用）
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-Bit TSS (Busy)	16 位 TSS（忙）
4	0	1	0	0	16-Bit Call Gate	16 位调用门
5	0	1	0	1	Task Gate	任务门
6	0	1	1	0	16-Bit Interrupt Gate	16 位中断门
7	0	1	1	1	16-Bit Trap Gate	16 位陷阱门
8	1	0	0	0	Reserved	保留
9	1	0	0	1	32-Bit TSS (Available)	32 位 TSS（可用）
10	1	0	1	0	Reserved	保留
11	1	0	1	1	32-Bit TSS (Busy)	32 位 TSS（忙）
12	1	1	0	0	32-Bit Call gate	32 位调用门
13	1	1	0	1	Reserved	保留
14	1	1	1	0	32-Bit Interrupt Gate	32 位中断门
15	1	1	1	1	32-Bit Trap Gate	32 位陷阱门