report

本报告已设置目录，使用 github 作为图床进行图片的挂载，如果以 markdown 格式阅读，还请麻烦打开代理软件。

2. 保护模式内存管理

2.1 内存管理概览

IA-32 架构的内存管理分为 分段 和 分页 两个部分。

- 分段：分段机制用于隔离各个代码、数据和堆栈模块，从而支持多个程序（或任务）在同一处理器上运行而不会相互干扰。分段可以在受保护的地址空间中隔离不同程序的执行内容。
- 分页：分页机制实现了一种传统的按需分页的虚拟内存系统，通过将程序的执行环境的某些部分按需映射到物理内存中。分页同样可以用于任务间的隔离。运行在保护模式下时，分段是必须的，而分页是可选的。

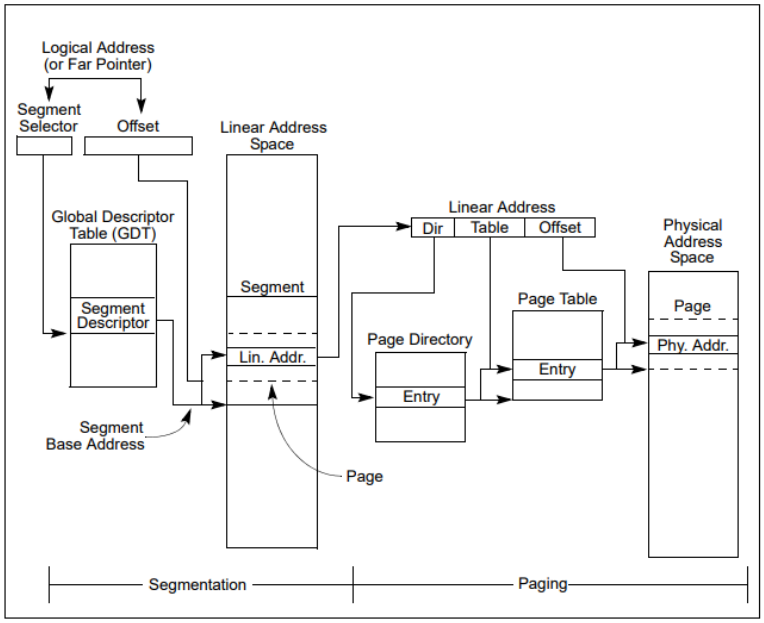
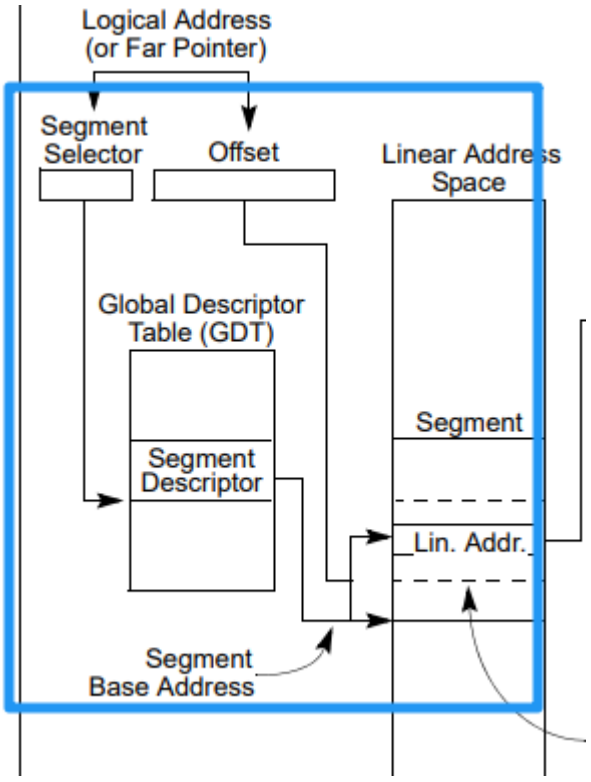


Figure 3-1. Segmentation and Paging

2.1.1 逻辑地址

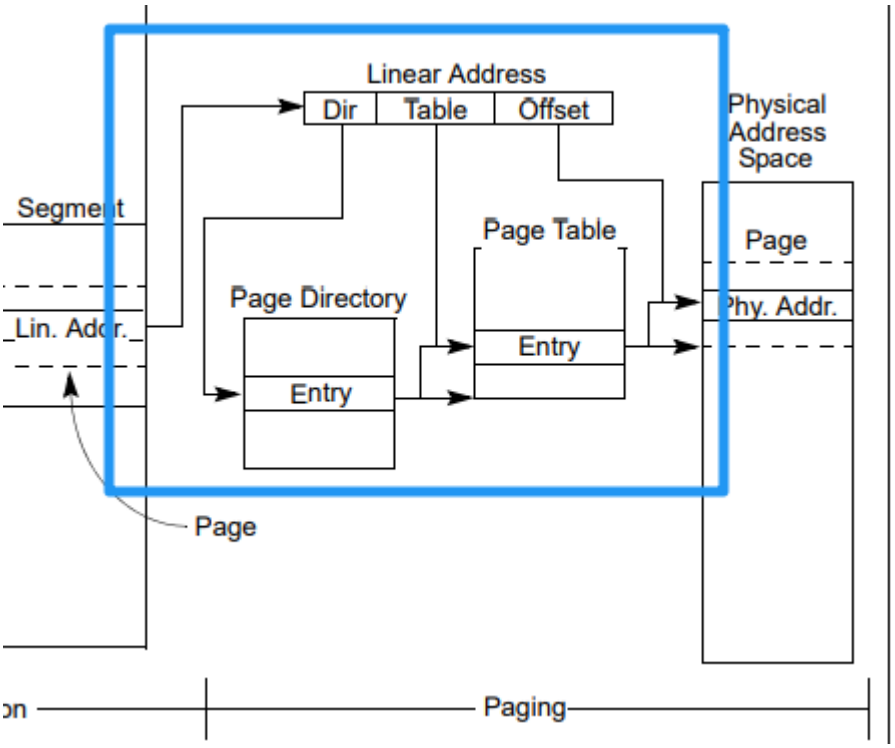
逻辑地址是程序员编写代码时所使用的地址。它由 段选择子（Segment Selector） 和 偏移量（Offset） 两部分组成。段选择子用于指定程序所需的段，而偏移量则定位段内的具体位置。逻辑地址在程序中并不会直接映射到物理内

存，而是需要经过段描述符转换成线性地址。下图是逻辑地址转换到线性地址的过程。



2.1.2 线性地址

线性地址是通过分段机制从逻辑地址转换而来的。每个段在全局描述符表（GDT）或局部描述符表（LDT）中都有对应的 段描述符（Segment Descriptor），该描述符记录了段的大小、访问权限和基址。处理器利用逻辑地址中的段选择子，从描述符表中查找段描述符并获取段的基址，将基址与偏移量相加后得到线性地址。下图是线性地址使用分页机制转换到物理地址的过程。



2.1.3 物理地址

物理地址是最终用于访问物理内存的地址。在线性地址基础上，可以通过分页机制进一步转换为物理地址。在没有启用分页的情况下，线性地址直接映射到物理地址。然而，如果启用分页，处理器会使用页表和页目录，将线性地址转换为物理地址。

2.1.4 分段机制的作用和实现

分段机制是 IA-32 架构中一种重要的内存保护方式，旨在实现代码、数据和堆栈的隔离，避免程序间的干扰。通过分段机制，不同的程序可以拥有独立的段集，处理器负责检查段边界，防止超越段界限的访问。

1. 段选择与转换过程：每个段在内存中都有一个段描述符，它存储了该段的大小、访问权限和基址。逻辑地址的段选择子指向段描述符位置，处理器根据描述符的基址和逻辑地址中的偏移量，计算出线性地址。
2. 分段类型与权限控制：分段机制不仅实现了段间隔离，还可以限制不同段的访问权限。例如，代码段只能执行而不可写入，数据段可读写但不可执行。这种控制增强了系统的安全性。

2.1.5 分页机制的简单介绍

分页是虚拟内存管理的核心机制，支持将较大的线性地址空间虚拟化为较小的物理内存。分页机制会在物理内存和磁盘之间交换数据，以满足程序对内存空间的需求。分页主要通过页目录和页表进行地址转换，将线性地址最终映射到物理地址。

2.2 分段机制

IA-32 架构支持的分段机制可以应用于多种系统设计，这些设计从简单的Basic Flat Model（仅对程序提供最小的保护）到Multi-Segment Model（通过分段创建一个稳健的操作环境，确保多个程序和任务可靠执行）不等。

2.2.1 Basic Flat Model（基本扁平模型）

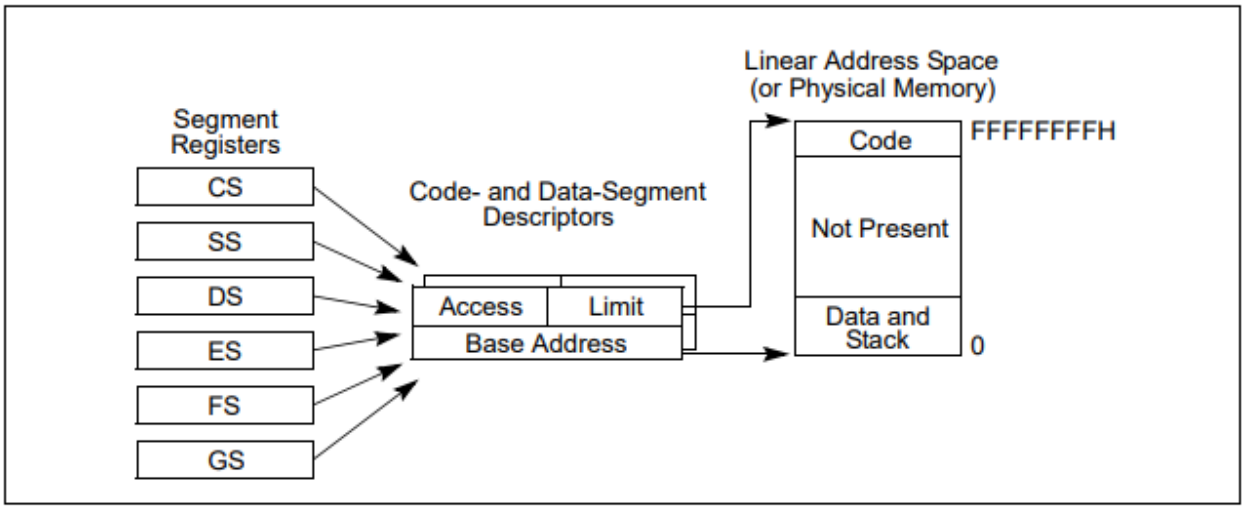


Figure 3-2. Flat Model

Basic Flat Model是最简单的内存模型，其中操作系统和应用程序共享一个连续的、未分段的地址空间。该模型尽可能地隐藏了架构的分段机制，使系统设计者和应用程序开发人员不需要直接接触分段。

在 IA-32 架构中实现基本扁平模型时，需要创建至少两个段描述符：一个用于代码段，一个用于数据段（参见图 3-2）。这两个段都映射到整个线性地址空间，这意味着它们的基地址为 0，段限长为 4 GB。这种设置确保即使在没有物理内存的地址上进行访问，分段机制也不会生成*内存引用溢出*的异常。(因为这个内存是虚拟的吗？请看下面的引用)

- 段限制 (Segment Limit)：设为 4 GB，以避免在无物理内存位置的访问时触发异常。
- 物理内存分布：通常将只读存储器（ROM）放置在物理地址空间的顶端，因为处理器在启动时从地址 `FFFF_FFF0H` 开始执行。随机存取存储器（RAM）则放置在地址空间的底部，因为在复位初始化后，数据段（DS）的初始基地址为 0。

> [https://www.cnblogs.com/tcicy/p/10185353.html] 当 CPU 运行于 32 位模式时，不管怎样，寄存器和指令都可以寻址整个线性地址空间，所以根本就不需要再去使用基地址或其他什么鬼东西。那为什么不干脆将基地址设成 0，好让逻辑

> 地址与线性地址一致呢？Intel 的文档将之称为 " 扁平模型 "（flat model），而且在现代的 x86 系统内核中就是这么做的（特别指出，它们使用的是基本扁平模型）。基本扁平模型（basic flat model）等价于在转换地址时关闭了分段功能。

2.2.2 Protected Flat Model（受保护的扁平模型）

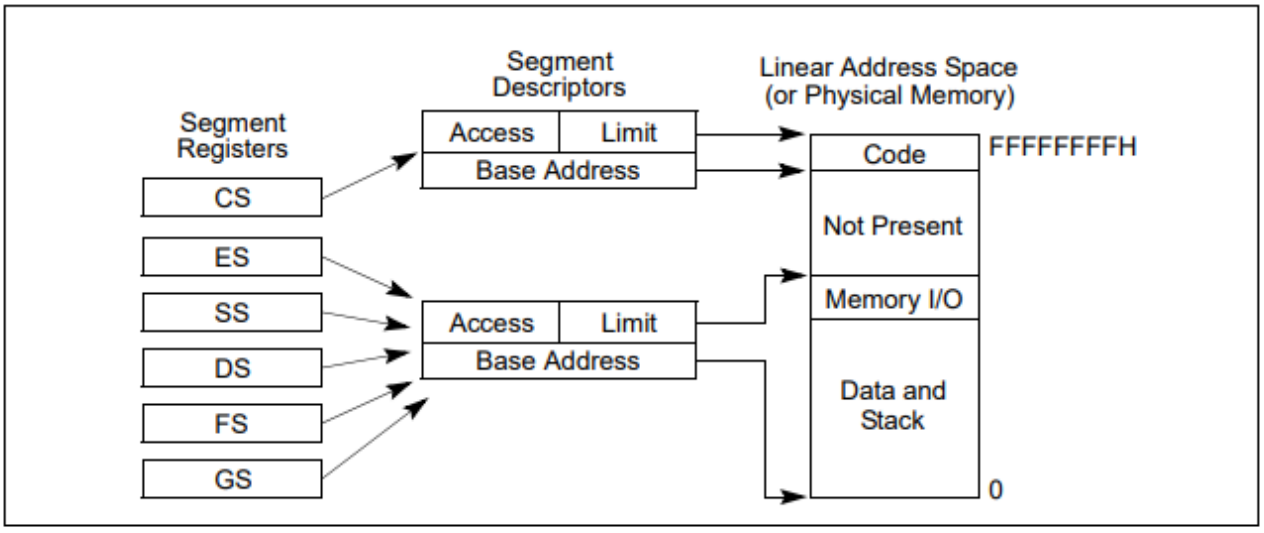


Figure 3-3. Protected Flat Model

受保护的扁平模型与基本扁平模型类似，但有一个关键区别：在该模型中，*段限制被设置为仅包含物理内存实际存在的地址范围*（如图 3-3 所示）。这样，当程序试图访问不存在的内存时，处理器会生成一般保护异常（#GP），提供了一种最小的硬件保护，能够防止某些类型的程序错误。

在此基础上，可以通过增加一些复杂性来实现更高级别的保护。例如，若要利用分页机制在用户代码/数据和管理代码/数据之间提供隔离，则需要定义四个段：

- 用户态的代码段和数据段，权限级别为 3（低权限）。
- 管理态的代码段和数据段，权限级别为 0（高权限）。

这些段可能会重叠（共享同一个地址空间），从线性地址空间的地址 0 开始。这个扁平的分段模型加上简单的分页结构，可以保护操作系统不受应用程序的影响。此外，通过为每个任务或进程提供独立的分页结构，还可以实现应用程序之间的隔离保护。

类似的设计被许多流行的多任务操作系统所采用，用于在用户和系统级别之间提供隔离保护，同时允许每个进程拥有自己的内存空间。

2.2.3 Multi-Segment Model（多段模型）

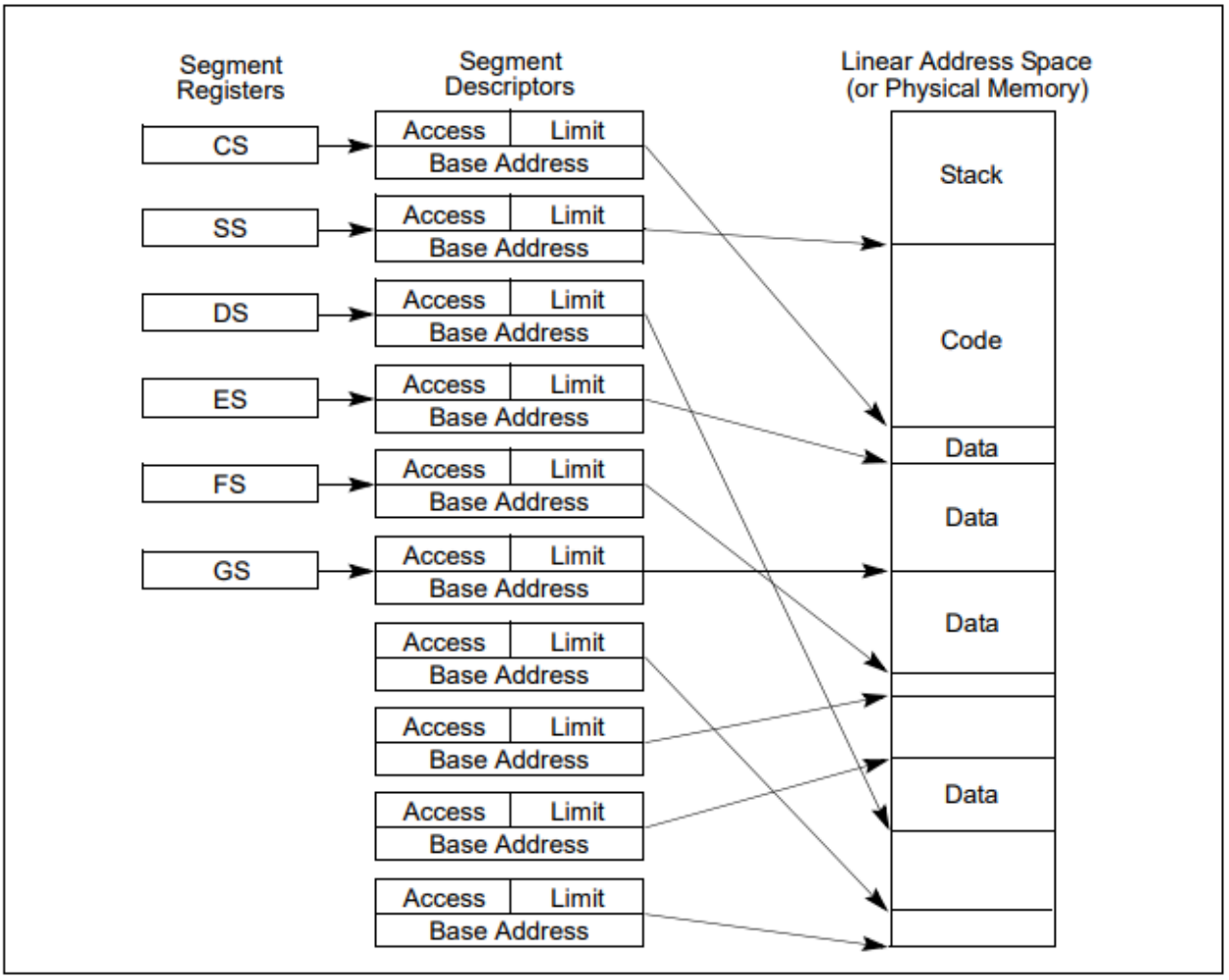


Figure 3-4. Multi-Segment Model

多段模型（如图 3-4 所示）充分利用了分段机制的全部功能，实现了代码、数据结构、程序和任务的硬件保护。在这种模型中，每个程序（或任务）都有自己的段描述符表和独立的段。这些段可以是完全私有的，仅供分配给它们的程序使用，也可以在多个程序之间共享。对所有段的访问以及系统上各程序的执行环境都受到硬件的控制。

访问控制不仅可以用于保护不越界访问段内地址，还可以防止在某些段内执行不允许的操作。例如，代码段被设定为只读段，硬件可以防止写入代码段。此外，通过为段创建的访问权限信息，还可以设置保护环（或称权限级别），用于限制对操作系统功能的未授权访问，确保应用程序无法直接操作系统级代码。

多段模型的主要特点

1. 隔离与共享：每个任务或程序拥有自己的段表，允许它们的段私有化，确保程序之间互不干扰。同时，必要时也可以实现段的共享。
2. 访问控制：段描述符包含的访问权限信息，可以实现对不同段的访问限制。例如，代码段为只读，数据段可读写，而堆栈段则只能用于特定的堆栈操作。

3. 权限级别：通过段的权限控制机制，可以实现系统的多级保护。操作系统可以在高权限级别运行，而应用程序则运行在低权限级别，从而防止应用程序对系统资源的未经授权访问。

2.3 逻辑地址和线性地址的转换

逻辑地址和线性地址的介绍我们在上面的内容已经提到过，我们先来解决一些概念上的问题，稍后再来探讨逻辑地址具体如何转换成了线性地址，以及处理器的行为。

2.3.1 段选择子 Segment Selectors

和 `report01` 中提到的段选择器是一个概念。

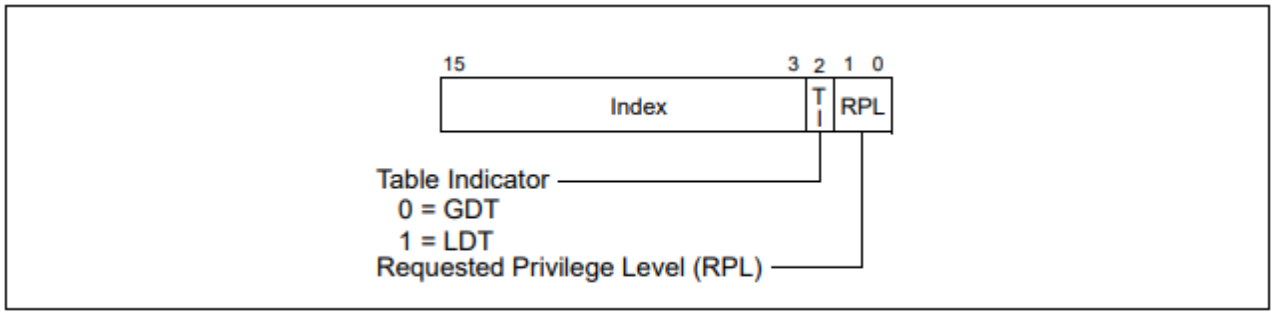


Figure 3-6. Segment Selector

段选择子是用于标识段的 16 位标识符（见图 3-6）。它并不直接指向段本身，而是指向定义该段的段描述符。段选择子包含以下几部分：

1. 索引（Index）（第 3 到 15 位）：用于选择全局描述符表（GDT）或局部描述符表（LDT）中的一个描述符。处理器将索引值乘以 8（因为每个段描述符占 8 字节），然后将结果加到 GDT 或 LDT 的基地址上（基地址分别存储在 GDTR 或 LDTR 寄存器中），从而定位段描述符。
2. 表指示符（TI）标志（第 2 位）：指定要使用的描述符表。清除此标志表示选择 GDT，设置此标志表示选择当前的 LDT。
3. 请求特权级（RPL）（第 0 和 1 位）：指定选择子的特权级，范围为 0 到 3，其中 0 是最高特权级。

特殊情况

- 空段选择子：GDT 的第一个入口（索引为 0，且 TI 标志为 0）不会被处理器使用，指向此入口的段选择子称为 " 空段选择子 "。当段寄存器（CS 或 SS 寄存器除外）加载了空选择子后，如果试图访问内存，则处理器会生成异常。

因此，空选择子可以用于初始化未使用的段寄存器。如果将 CS 或 SS 寄存器加载为空选择子，则会生成一般保护异常（#GP）。

段选择子在应用程序中作为指针变量的一部分是可见的，但通常由链接编辑器或链接加载器分配或修改，而不是由应用程序本身直接操作。

2.3.2 段寄存器 Segment Registers

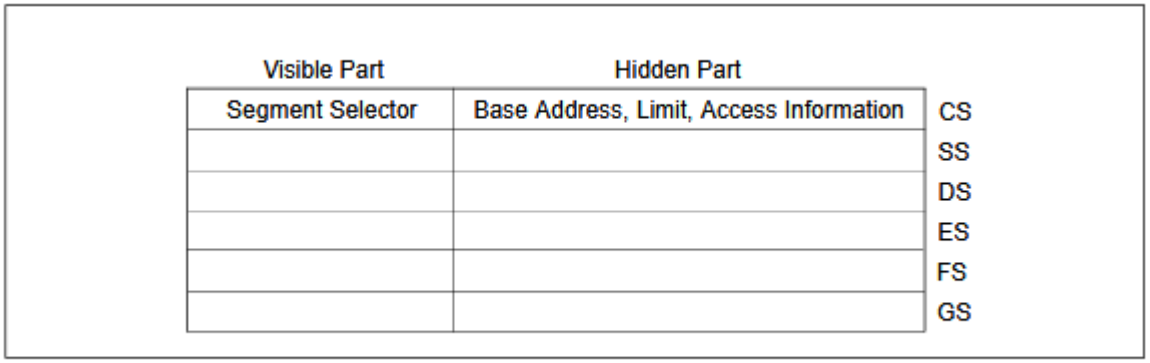


Figure 3-7. Segment Registers

为了减少地址转换时间和简化编码，处理器提供了用于存储最多 6 个段选择子的寄存器（见图 3-7）。这些段寄存器各自支持特定类型的内存引用（代码、堆栈或数据）。要进行任何程序执行，至少需要代码段（CS）、数据段（DS）和堆栈段（SS）寄存器被加载有效的段选择子。此外，处理器还提供了三个额外的数据段寄存器（ES、FS 和 GS），可以使当前执行的程序（或任务）能够访问更多的数据段。

为了让程序访问一个段，该段的段选择子必须加载到其中一个段寄存器中。因此，尽管系统可以定义成千上万个段，但在任何时刻只有 6 个段可供立即使用。其他段可以通过在程序执行过程中将它们的段选择子加载到这些寄存器来获得访问权限。

可见部分和隐藏部分

每个段寄存器都有一个 " 可见部分 " 和一个 " 隐藏部分 "。当一个段选择子加载到段寄存器的可见部分时，处理器会将该段选择子所指向的段描述符中的基地址、段限长和访问控制信息加载到*隐藏部分*中。寄存器中缓存的信息（可见部分和隐藏部分）允许处理器在不占用额外总线周期读取段描述符的情况下直接进行地址转换。

在多处理器系统中，如果多个处理器访问同一描述符表，当描述符表被修改时，软件有责任重新加载段寄存器。如果不这样做，则在描述符表的内存驻留版本被修改后，段寄存器中缓存的旧段描述符可能会被继续使用。

加载段寄存器的指令

有两种加载段寄存器的指令：

1. 直接加载指令：如 `MOV`、`POP`、`LDS`、`LES`、`LSS`、`LGS` 和 `LFS` 指令，这些指令显式引用段寄存器。
2. 隐式加载指令：如远指针版本的 `CALL`、`JMP` 和 `RET` 指令，`SYSENTER` 和 `SYSEXIT` 指令，以及 `IRET`、`INTn`、`INTO` 和 `INT3` 指令。这些指令在操作过程中会改变 `CS` 寄存器（有时也会改变其他段寄存器）的内容。

此外，`MOV` 指令也可以用于将段寄存器的可见部分存储到通用寄存器中。

2.3.3 段描述子 Segment Descriptors

主要是掌握段描述子的结构。

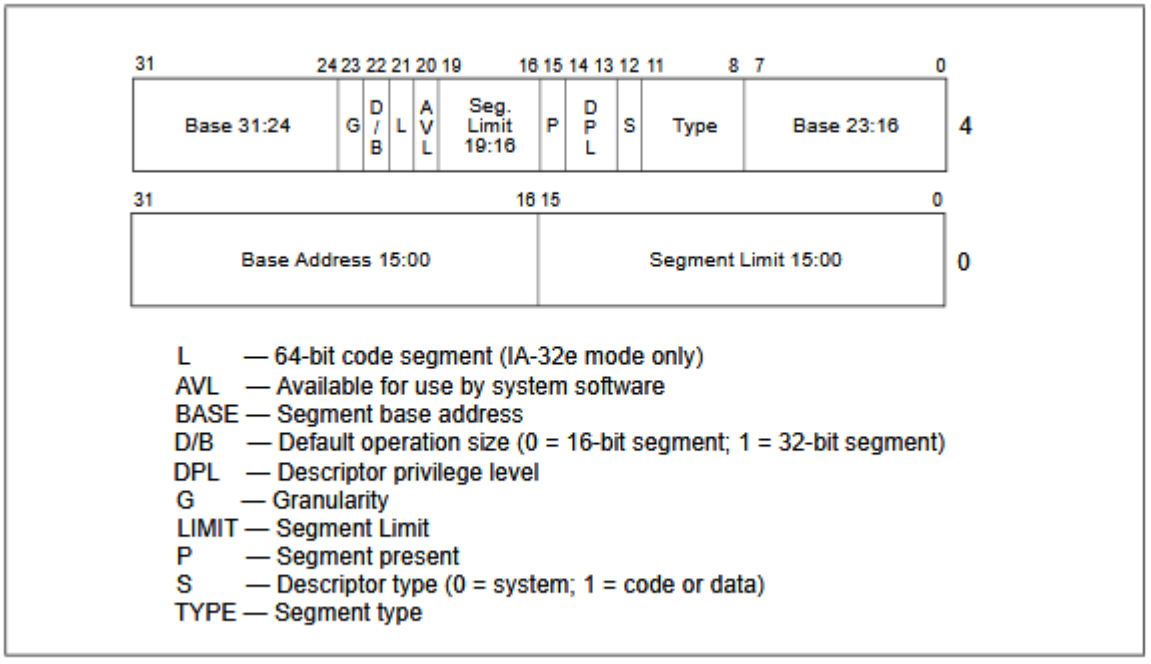


Figure 3-8. Segment Descriptor

段描述符是一种存储在全局描述符表（GDT） 或 局部描述符表（LDT） 中的数据结构，为处理器提供段的大小、位置、访问控制和状态信息。段描述符通常由编译器、链接器、加载器或操作系统创建，而不是由应用程序直接生成。图 3-8 展示了所有类型的段描述符的通用格式。

字段说明

1. Base Address（基地址）：定义了段的第 0 字节在 4 千兆字节线性地址空间中的位置。处理器将三个基地址字段组合成一个单一的 32 位值。基地址字段

分为三个部分 `Base 31:24`、`Base 23:16` 和 `Base 15:0`，共同定义了段的起始地址。

2. Segment Limit（段限长）：段限长也分为两部分，`Limit 19:16` 和 `Limit 15:0`，处理器将两个段限值字段组合成一个 20 位的值，定义了段的大小。当段粒度位（G）设为 0 时，单位为字节；设为 1 时，单位为 4 KB。
3. Type（类型）：定义段的类型。类型字段用于区分代码段、数据段和系统段，并指定特定的访问权限。
4. S（描述符类型）：指示描述符的类型，0 表示系统段，1 表示代码或数据段。
5. DPL（描述符特权级）：定义段的特权级，范围从 0 到 3，数值越低权限越高。处理器利用 DPL 来实现内存访问的安全控制。
6. P（段存在位）：表示段是否存在于内存中。若 P 位为 0，表示该段当前不在内存中，访问该段将导致段不存在异常。
7. D/B（默认操作大小）：决定段的默认操作模式。0 表示 16 位段，1 表示 32 位段。
8. G（粒度）：设置段限长的单位。G 位为 0 表示段限长单位为字节，G 位为 1 表示段限长单位为 4 KB。
9. L（64 位代码段）：仅在 IA-32e 模式下使用，指示该段是否为 64 位代码段。
10. AVL（系统软件可用位）：为系统软件保留使用，通常由操作系统管理。

2.3.4 地址转换过程&CPU 行为

逻辑地址转换为线性地址

- 获取段描述符：
  - 处理器根据逻辑地址中的段选择子定位全局描述符表（GDT）或局部描述符表（LDT），找到对应的段描述符。
  - 段选择子包含一个索引值，用于指向 GDT 或 LDT 中的特定段描述符。处理器将索引乘以 8（每个段描述符占 8 字节），并将结果加上 GDT 或 LDT 的基地址，来确定段描述符的位置。
- 加载段描述符信息：
  - 处理器将段描述符的基地址、段限长和访问控制信息加载到段寄存器的 " 隐藏部分 "（也称为描述符缓存）。这一步骤确保了处理器可以快速访问段的基地址和限制信息，无需每次访问都读取段描述符。
- 检查访问权限和边界：
  - 处理器检查段描述符中的访问权限（特权级、类型等），以确保当前特权级允许访问该段。同时，它还会检查逻辑地址中的偏移量是否在段限

长范围内。如果偏移量超出段限长，处理器将生成 " 超出段限长异常 "。

- 计算线性地址：
  - 如果访问权限和偏移量都有效，处理器将段描述符中的基地址与逻辑地址中的偏移量相加，生成线性地址。
  - 这个线性地址位于处理器的线性地址空间中，是一个统一的地址表示，可以进一步转换为物理地址。

CPU 做了什么？

在逻辑地址转换为线性地址的过程中，处理器完成了以下四件事：

1. 定位段描述符。
2. 加载段描述符中的基地址、段限长和访问控制信息。
3. 检查访问权限和段边界。
4. 将基地址与偏移量相加，生成线性地址。

2.4 描述符的分类

主要关于段描述符、段描述符表和门描述符的介绍，其中后两者被归为系统描述符。

2.4.1 段描述符分类

Code and Data Segment Descriptor

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

当段描述符中的 S 标志（描述符类型）为 1 时，描述符为代码段描述符或者数据段描述符。类型域的最高位（段描述符的第二个双字的第 11 位）将决定该描述符为数据段描述符（为 0）还是代码段描述符（为 1）。对于数据段而言，描述符的类型域的低 3 位（第 8、9、10 位）被解释为访问控制（A）、是否可写（W）、扩展方向（E）。

堆栈段必须是可读写的数据段。将一个不可写的数据段的选择子置入 SS 寄存器会导致一般保护异常（#GP）。如果堆栈段的大小需要动态变化，可以将其置为向下扩展数据段（扩展方向标志为 1）。这里，动态改变段界限将导致栈空间朝着栈底部空间扩展。如果段的长度保持不变，堆栈段可以是向上扩展的，也可以是向下扩展的。对于代码段而言，类型域的低 3 位被解释为访问位（A）、可读位（R）、一致位（C）。访问位表示自最后一次被操作系统清零后，段是否被访问过。每当处理器将段的段选择子置入段寄存器时，就将访问位置为 1。该位一直保持为 1 直到被显式清零。该位可以用于虚拟内存管理和调试。代码段可以是一致性的，也可以是不一致性的。转入一个特权级更高的一致性段的进程可以在当前特权级继续执行下去。除非使用了调用门或者任务门，否则，转入一个不同特权级的非一致性段将使处理器产生一个 " 一般保护异常 "（#GP）。不访问受保护程序的系统程序和某些类型的异常处理程序（比如除法错或者溢出）可以被载入一致性的代码段。不能被特权级更低的程序和过程访问的程序应该被载入非一致性的代码段。

无论目标段是否为一致性代码段，*进程都不能因为 call 或 jump 而转入一个特权级较低（特权值较大）的代码段执行*。试图进行这样的执行转换将导致一个一般保护异常（#GP）。

所有的数据段都是非一致性的，*这就意味着数据段不能被更低特权级的进程访问（特权级数值较大的执行代码）*。然而，和代码段不同，数据段可以被更高优先级的程序或者过程（特权级数值较小的执行代码）访问，不需要使用特别的访问门。

系统描述符

当段描述符中的 S 标志（描述符类型）被清除时，描述符类型为系统描述符。处理器可以识别以下几种系统描述符类型：

- 局部描述符表（LDT）段描述符
- 任务状态段（TSS）描述符
- 调用门（Call Gate）描述符
- 中断门（Interrupt Gate）描述符
- 陷阱门（Trap Gate）描述符
- 任务门（Task Gate）描述符

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（例如 LDT 和 TSS 段）。
- 门描述符：本身充当 " 门 "，包含指向代码段中过程入口点的指针（如调用门、中断门和陷阱门），或包含 TSS 的段选择子（如任务门）。

表 3-2 展示了系统段描述符和门描述符的类型字段编码。在 IA-32e 模式下，系统描述符为 16 字节，而不是 8 字节。

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

段描述符表

段描述符表是一个段描述符的数组。段描述符表的长度是可变的，最多可以包含 8192（2^{13}）个 8 字节的描述符。有两种描述符表：

- 全局描述符表（GDT）
- 局部描述符表（LDT）。

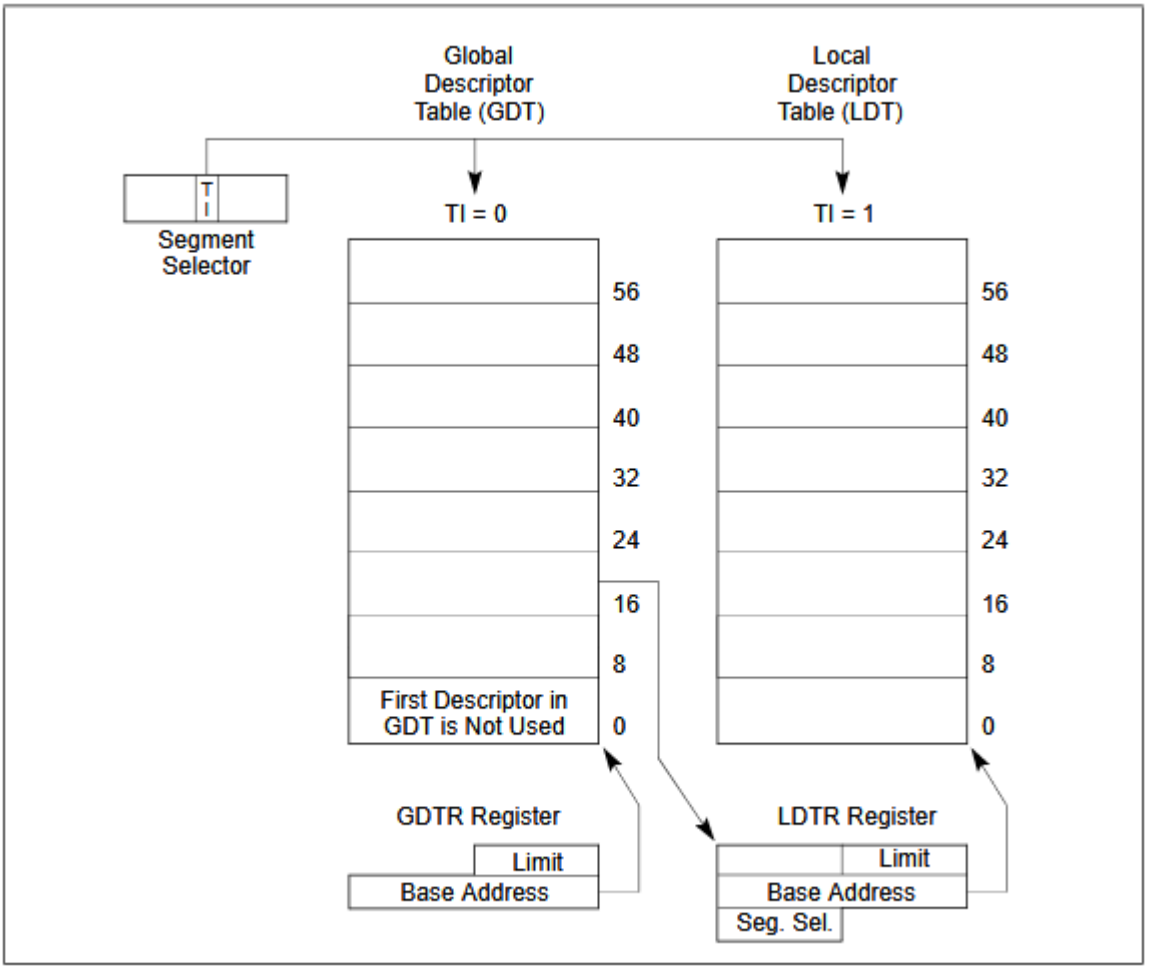


Figure 3-10. Global and Local Descriptor Tables

系统必须定义一个 GDT，以供所有的进程或者任务使用。也可以定义一个或者多个 LDT。比如，可以为每个正在运行的任务定义一个 LDT,也可以让所有的任务共享一个 LDT。

GDP 的特性

GDT 本身不是一个段，而是线性地址空间中的一个数据结构。GDT 的线性基地址和界限必须被装入 GDTR 寄存器。

GDT 的基地址应当按照 8 字节方式对齐，这样可以获得最好的处理器性能。GDT 的界限是按字节计算的。在段中，段界限加上段基地址就可以得到段最后一个字节的有效地址。0 界限值表示只有一个有效的字节。因为段描述符总是 8 字节长，GDT 的界限应该总是 8 的整数倍减 1（即 8N－1）。

空描述符

处理器并不使用 GDT 中的第一个描述符。当指向这个 NULL 描述符的段选择子被装入数据段寄存器（DS、ES、FS 或者 GS）时，处理器并不产生异常。 但是如果使用这个 NULL 描述符来访问内存，处理器就会产生一个一般保护异常（`#GP`）。使用这个指向 NULL 描述符的段选择子来初始化段寄存器，这样可以确保在不经意引用未使用的段寄存器时，处理器能产生一个异常。

LDP 的特性

LDT 位于类型为 LDT 的系统段内。GDT 必须包含一个指向 LDT 段的段描述符。如果系统支持多个 LDT，那么每个 LDT 段都要有一个段选择子，都要在 GDT 中有一个段描述符。每个 LDT 必须在 GDT 中拥有独立的段描述符。LDT 的段描述符可以位于 GDT 中的任何地方。LDT 是通过它的段选择子来访问的。为了避免在访问 LDT 时进行地址翻译，LDT 的段选择子、线性基地址、段界限和访问权限都放在 LDTR 寄存器中。

LDP 的访问

1. GDT 中的 LDT 段描述符：在系统初始化时，GDT 中包含一个或多个 LDT 的段描述符。该段描述符定义了 LDT 的基地址、限长和访问权限。
2. 加载到 LDTR 寄存器：当需要访问某个 LDT 时，系统会将 LDT 的段选择子加载到 LDTR 寄存器中。此时，处理器会自动从 GDT 中读取该段选择子指向的 LDT 段描述符信息，并将 LDT 的基地址、限长和访问权限加载到 LDTR 寄存器的隐藏部分中。
3. 使用 LDTR 寄存器进行访问：一旦 LDT 的基地址和限长信息存储在 LDTR 寄存器中，处理器便可以直接通过 LDTR 寄存器访问 LDT，而无需再次访问 GDT。这种机制有助于提高访问 LDT 的效率，因为缓存信息减少了地址转换的开销。