CHAPTER 3 PROTECTED-MODE MEMORY MANAGEMENT

本章描述Intel 64和IA-32架构的保护模式内存管理设施，包括物理内存要求、分段机制和分页机制。

另请参阅：第5章“保护”（关于处理器保护机制的描述）和第20章“8086仿真”（关于实地址和虚拟8086模式下内存寻址保护的描述）。

3.1 MEMORY MANAGEMENT OVERVIEW

IA-32架构的内存管理设施分为两部分：分段和分页。分段提供了一种隔离独立代码、数据和堆栈模块的机制，使得多个程序（或任务）可以在同一处理器上运行而互不干扰。分页提供了一种实现传统请求分页虚拟内存系统的机制，其中程序的执行环境部分会根据需要映射到物理内存中。分页也可用于提供多个任务之间的隔离。在保护模式下运行时，必须使用某种形式的分段。没有模式位可以禁用分段。然而，分页的使用是可选的。

这两种机制（分段和分页）可被配置用于支持简单的单程序（或单任务）系统、多任务系统，或采用共享内存的多处理器系统。

如图3-1所示，分段机制提供了一种将处理器的可寻址内存空间（称为线性地址空间）划分为更小的受保护地址空间（称为段）的方法。段可用于存储程序的代码、数据和堆栈，或保存系统数据结构（如TSS或LDT）。如果处理器上运行多个程序（或任务），可以为每个程序分配自己的一组段。处理器会强制维护这些段之间的边界，并确保一个程序不会通过写入另一个程序的段来干扰其执行。分段机制还允许对段进行类型划分，从而限制对特定类型段可执行的操作。

系统中的所有段都包含在处理器线性地址空间中。要定位特定段中的字节，必须提供逻辑地址（也称为远指针）。逻辑地址由段选择子和偏移量组成。段选择子是段的唯一标识符，除其他功能外，它还提供指向描述符表（如全局描述符表GDT）中称为段描述符的数据结构的偏移量。每个段都有一个段描述符，用于指定段的大小、段的访问权限和特权级、段类型以及段第一个字节在线性地址空间中的位置（称为段的基地址）。逻辑地址的偏移量部分与段的基地址相加，以定位段内的字节。因此，基地址加上偏移量形成了处理器线性地址空间中的线性地址。

CHAPTER 3 PROTECTED-MODE MEMORY MANAGEMENT

This chapter describes the Intel 64 and IA-32 architecture’s protected-mode memory management facilities, including the physical memory requirements, segmentation mechanism, and paging mechanism.

See also: Chapter 5, “Protection” (for a description of the processor’s protection mechanism) and Chapter 20, “8086 Emulation” (for a description of memory addressing protection in real-address and virtual-8086 modes).

3.1 MEMORY MANAGEMENT OVERVIEW

The memory management facilities of the IA-32 architecture are divided into two parts: segmentation and paging. Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another. Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program’s execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks. When operating in protected mode, some form of segmentation must be used. **There is no mode bit to disable segmentation.** The use of paging, however, is optional.

These two mechanisms (segmentation and paging) can be configured to support simple single-program (or single-task) systems, multitasking systems, or multiple-processor systems that used shared memory.

As shown in Figure 3-1, segmentation provides a mechanism for dividing the processor’s addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**. Segments can be used to hold the code, data, and stack for a program or to hold system data structures (such as a TSS or LDT). If more than one program (or task) is running on a processor, each program can be assigned its own set of segments. The processor then enforces the boundaries between these segments and insures that one program does not interfere with the execution of another program by writing into the other program’s segments. The segmentation mechanism also allows typing of segments so that the operations that may be performed on a particular type of segment can be restricted.

All the segments in a system are contained in the processor’s linear address space. To locate a byte in a particular segment, a **logical address** (also called a far pointer) must be provided. A logical address consists of a segment selector and an offset. The segment selector is a unique identifier for a segment. Among other things it provides an offset into a descriptor table (such as the global descriptor table, GDT) to a data structure called a segment descriptor. Each segment has a segment descriptor, which specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a **linear address** in the processor’s linear address space.

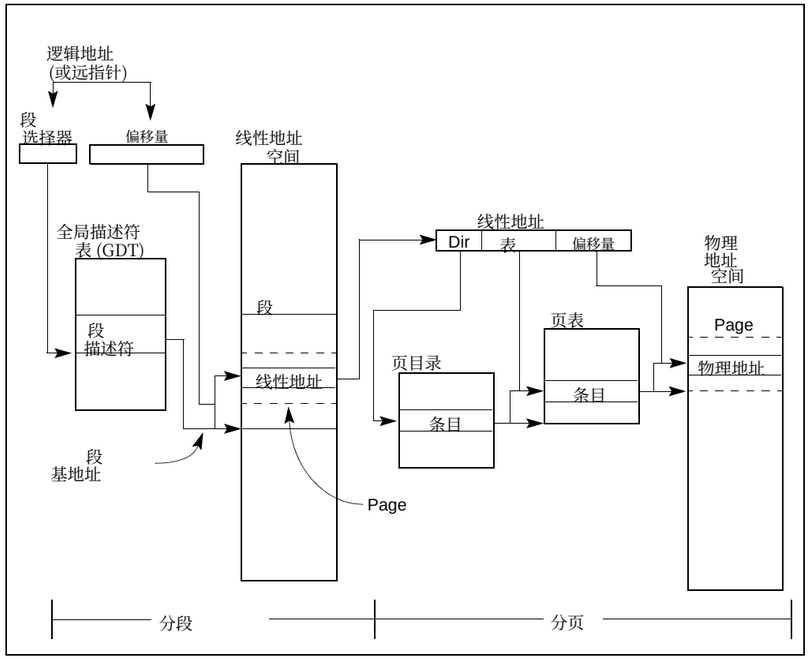


图3-1. 分段与分页

若不使用分页，处理器的线性地址空间将直接映射到处理器的物理地址空间。物理地址空间定义为处理器在其地址总线上可生成的地址范围。

由于多任务计算系统通常定义的线性地址空间远大于经济上可行的一次性装入物理内存的容量，因此需要某种“虚拟化”线性地址空间的方法。这种线性地址空间的虚拟化通过处理器的分页机制来处理。

分页支持一种“虚拟内存”环境，其中使用少量物理内存（RAM和ROM）及部分磁盘存储来模拟大型线性地址空间。使用分页时，每个段被划分为页（通常每页大小为4 KB），这些页存储在物理内存或磁盘上。操作系统或执行体维护一个页目录和一组页表来跟踪这些页。当程序（或任务）尝试访问线性地址空间中的某个地址位置时，处理器使用页目录和页表将线性地址转换为物理地址，然后对内存位置执行请求的操作（读取或写入）。

如果被访问的页当前不在物理内存中，处理器会中断程序的执行（通过产生页错误异常）。随后操作系统或执行体从磁盘将该页读入物理内存，并继续执行程序。

当在操作系统或执行体中正确实现分页时，物理内存与磁盘之间的页交换对程序的正确执行是透明的。即使是为16位IA-32处理器编写的程序，在虚拟8086模式下运行时也能（透明地）进行分页。

3.2 使用段

IA-32架构支持的分段机制可用于实现多种系统设计。这些设计范围从仅最低限度使用分段来保护

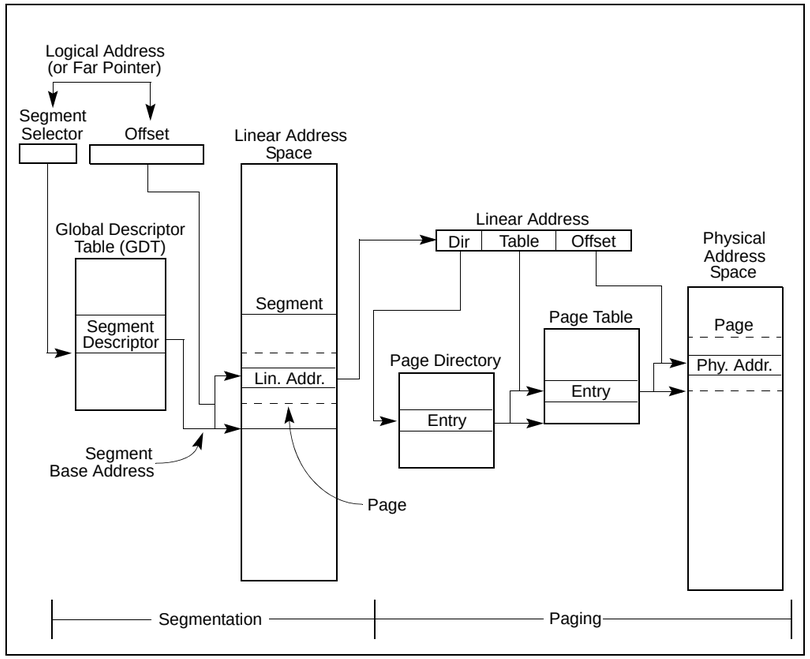


Figure 3-1. Segmentation and Paging

If paging is not used, the linear address space of the processor is mapped directly into the physical address space of processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of “virtualizing” the linear address space is needed. This virtualization of the linear address space is handled through the processor’s paging mechanism.

Paging supports a “virtual memory” environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk storage. When using paging, each segment is divided into pages (typically 4 KBytes each in size), which are stored either in physical memory or on the disk. The operating system or executive maintains a page directory and a set of page tables to keep track of the pages. When a program (or task) attempts to access an address location in the linear address space, the processor uses the page directory and page tables to translate the linear address into a physical address and then performs the requested operation (read or write) on the memory location.

If the page being accessed is not currently in physical memory, the processor interrupts execution of the program (by generating a page-fault exception). The operating system or executive then reads the page into physical memory from the disk and continues executing the program.

When paging is implemented properly in the operating-system or executive, the swapping of pages between physical memory and the disk is transparent to the correct execution of a program. Even programs written for 16-bit IA-32 processors can be paged (transparently) when they are run in virtual-8086 mode.

3.2 USING SEGMENTS

The segmentation mechanism supported by the IA-32 architecture can be used to implement a wide variety of system designs. These designs range from flat models that make only minimal use of segmentation to protect

程序的平坦模型，到采用分段创建健壮操作环境的多段模型——在该环境中多个程序和任务能够可靠地执行。

以下章节将提供若干示例，说明如何在系统中运用分段机制来提升内存管理的性能与可靠性。

3.2.1 基本平坦模型

系统中最简单的内存模型是基本的“平坦模型”，在该模型中，操作系统和应用程序可访问一个连续、未分段的地址空间。这种基本平坦模型尽可能地向系统设计者和应用程序员隐藏了架构的分段机制。

要在IA-32架构中实现基本平坦内存模型，至少需要创建两个段描述符：一个用于引用代码段，另一个用于引用数据段（参见图3-2）。然而，这两个段都被映射到整个线性地址空间：即两个段描述符具有相同的基地址值0和相同的段限长4 GB。通过将段限长设置为4 GB，即使特定地址没有物理内存存在，分段机制也不会因越界内存访问而产生异常。ROM（EPROM）通常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）则被放置在地址空间的底部，因为复位初始化后DS数据段的初始基地址为0。

3.2.2 受保护平坦模型

受保护平坦模型与基本平坦模型类似，不同之处在于段限长被设置为仅包含物理内存实际存在的地址范围（见图3-3）。任何访问不存在内存的尝试都会触发通用保护异常（#GP）。该模型提供了针对某些程序错误的最低级别硬件保护。

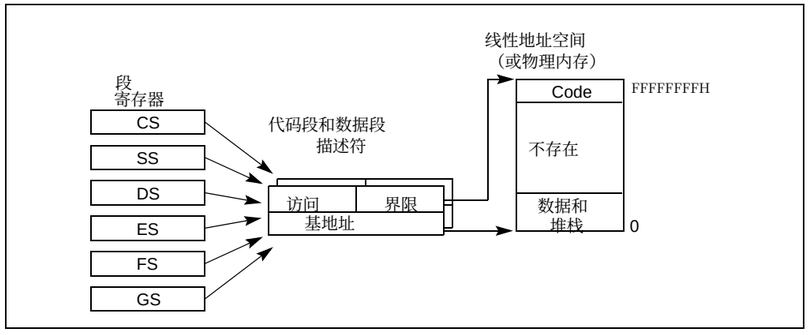


图 3-2 平坦模型

programs to multi-segmented models that employ segmentation to create a robust operating environment in which multiple programs and tasks can be executed reliably.

The following sections give several examples of how segmentation can be employed in a system to improve memory management performance and reliability.

3.2.1 Basic Flat Model

The simplest memory model for a system is the basic “flat model,” in which the operating system and application programs have access to a continuous, unsegmented address space. To the greatest extent possible, this basic flat model hides the segmentation mechanism of the architecture from both the system designer and the application programmer.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created, one for referencing a code segment and one for referencing a data segment (see Figure 3-2). Both of these segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes. By setting the segment limit to 4 GBytes, the segmentation mechanism is kept from generating exceptions for out of limit memory references, even if no physical memory resides at a particular address. ROM (EPROM) is generally located at the top of the physical address space, because the processor begins execution at FFFF_FFF0H. RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.

3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

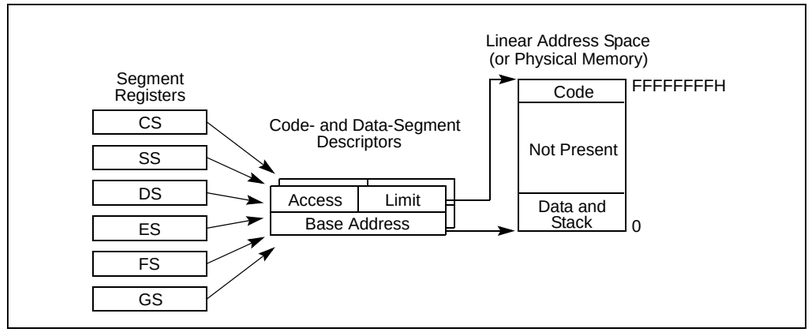


Figure 3-2. Flat Model

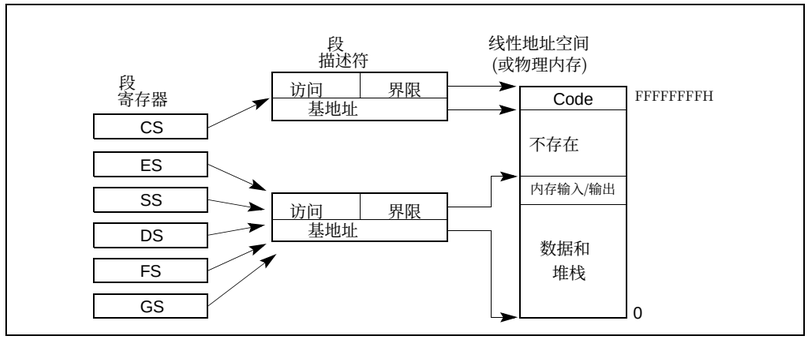


图3-3. 受保护平坦模型

可以为该受保护平坦模型增加更多复杂性以提供进一步保护。例如，若要通过分页机制实现用户与监管者代码及数据之间的隔离，需要定义四个段：特权级3的用户代码段和数据段，以及特权级0的监管者代码段和数据段。这些段通常相互重叠，并起始于线性地址空间中的地址0。这种平坦分段模型配合简单的分页结构可以保护操作系统免受应用程序影响，若为每个任务或进程添加独立的分页结构，还能实现应用程序间的相互保护。多个流行的多任务操作系统都采用了类似的设计方案。

3.2.3 多段模型

多段模型（如图3-4所示）利用分段机制的全部功能，为代码、数据结构和程序及任务提供硬件强制保护。在此模型中，每个程序（或任务）都有其自己的段描述符表和自己的段。这些段可以完全私有于其分配的程序，也可以在程序间共享。对系统中运行的所有段及单个程序执行环境的访问均由硬件控制。

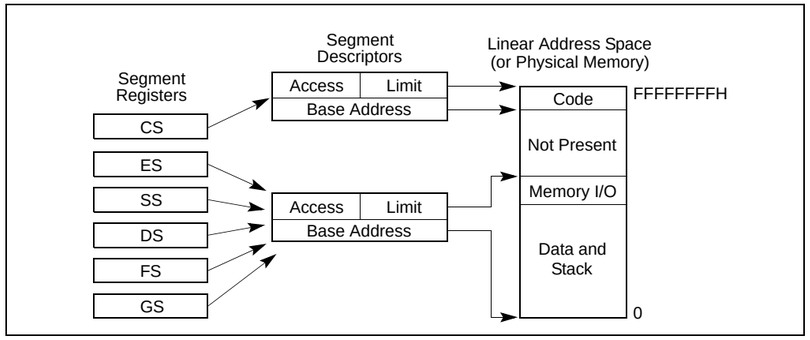


Figure 3-3. Protected Flat Model

More complexity can be added to this protected flat model to provide more protection. For example, for the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined: code and data segments at privilege level 3 for the user, and code and data segments at privilege level 0 for the supervisor. Usually these segments all overlay each other and start at address 0 in the linear address space. This flat segmentation model along with a simple paging structure can protect the operating system from applications, and by adding a separate paging structure for each task or process, it can also protect applications from each other. Similar designs are used by several popular multitasking operating systems.

3.2.3 Multi-Segment Model

A multi-segment model (such as the one shown in Figure 3-4) uses the full capabilities of the segmentation mechanism to provided hardware enforced protection of code, data structures, and programs and tasks. Here, each program (or task) is given its own table of segment descriptors and its own segments. The segments can be completely private to their assigned programs or shared among programs. Access to all segments and to the execution environments of individual programs running on the system is controlled by hardware.

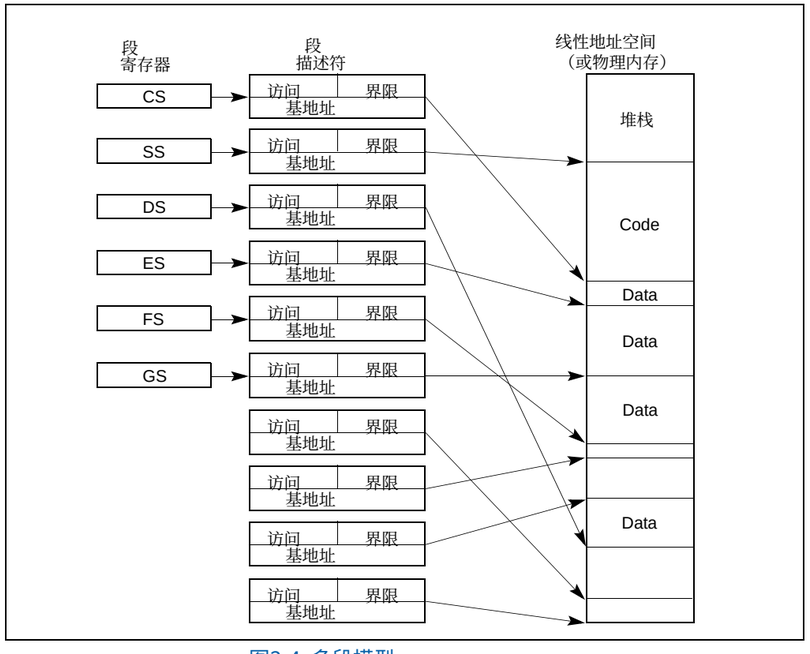


图3-4. 多段模型

访问检查不仅可用于防止引用段界限之外的地址，还可用于防止在特定段中执行不允许的操作。例如，由于代码段被指定为只读段，硬件可用于防止写入代码段。为段创建的访问权限信息也可用于设置保护环或保护级别。保护级别可用于保护操作系统过程免受应用程序的未经授权访问。

3.2.4 IA-32e模式下的分段机制

在Intel 64架构的IA-32e模式下，分段机制的效果取决于处理器是运行在兼容模式还是64位模式。在兼容模式下，分段功能与传统16位或32位保护模式语义完全相同。

在64位模式下，分段机制通常（但非完全）被禁用，从而创建一个平坦64位线性地址空间。处理器将CS、DS、ES、SS的段基址视为零，使得线性地址等于有效地址。FS和GS段属于例外情况。这些段寄存器（保存段基址）可作为线性地址计算中的附加基址寄存器使用，它们便于访问局部数据和某些操作系统数据结构。

请注意，处理器在64位模式下运行时不会执行段限检查。

3.2.5 分页与分段

分页机制可与图3-2、图3-3和图3-4所述的任何分段模型结合使用。处理器的分页机制将线性地址空间（段被映射至其中）划分为若干页（如图3-1所示）。这些线性地址空间的页随后被映射到物理地址空间中的页。分页机制提供了多种页面级保护机制，既可单独使用，也可与分段保护机制配合使用。

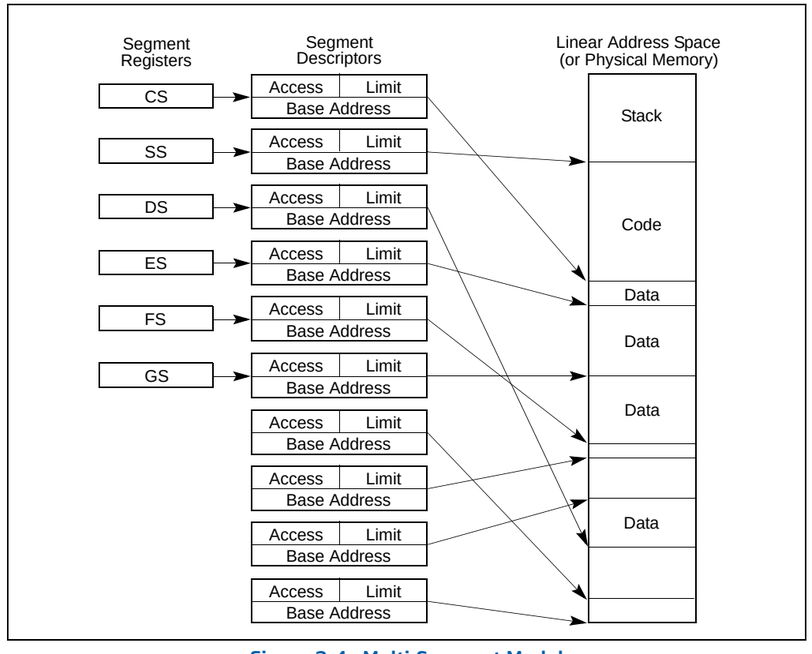


Figure 3-4. Multi-Segment Model

Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as an additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.

Note that the processor does not perform segment limit checks at runtime in 64-bit mode.

3.2.5 Paging and Segmentation

Paging can be used with any of the segmentation models described in Figures 3-2, 3-3, and 3-4. The processor's paging mechanism divides the linear address space (into which segments are mapped) into pages (as shown in Figure 3-1). These linear-address-space pages are then mapped to pages in the physical address space. The paging mechanism offers several page-level protection facilities that can be used with or instead of the segment-

例如，它允许以逐页为基础实施读写保护。该分页机制还提供两级用户-监管者保护，同样可按逐页方式指定。

3.3 物理地址空间

在保护模式下，IA-32架构提供了一个常规的4 GB（2字节）物理地址空间。这是处理器能够在其地址总线上寻址的地址空间。该地址空间是平坦的（未分段），地址范围从0连续延伸到FFFFFFFFH。此物理地址空间可映射到读写存储器、只读存储器和内存映射I/O。本章描述的内存映射机制可用于将此物理内存划分为段和/或页。

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2<style id='1'> </style>字节（64 GB），最大物理地址为FFFFFFFFFH。可通过以下两种方式之一启用此扩展功能：

- 使用使用位于控制寄存器CR的位5中的物理地址扩展 (PAE) 标志，4.
- 使用36位页面大小扩展 (PSE-36) 功能（在Pentium III 处理器中引入）。

物理地址支持此后已扩展到36位以上。有关36位物理寻址的更多信息，请参阅第4章“分页”。

3.3.1 Intel® 64 处理器与物理地址空间

在支持Intel 64 架构的处理器上（CPUID.80000001:EDX[29] = 1），物理地址范围的大小是具体实现相关的，并由CPUID.80000008H:EAX[的位7-0]指示。

有关EAX中返回信息的格式，请参阅《Intel® 64 和 IA-32 架构软件开发人员手册》第2A卷第3章中的“CPUID—CPU识别”。另请参阅：第4章“分页”。

3.4 逻辑地址与线性地址

在保护模式的系统架构级别，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。

即使以最小化方式使用段，处理器地址空间中的每个字节都是通过逻辑地址访问的。逻辑地址由16位段选择子和32位偏移量组成（见图3-5）。段选择子用于标识字节所在的段，偏移量则指定字节在段中相对于段基地址的位置。

处理器将每个逻辑地址转换为线性地址。线性地址是处理器线性地址空间中的32位地址。与物理地址空间类似，线性地址空间是一个平坦（未分段）的2^32字节地址空间，地址范围从0到FFFFFFFFH。线性地址空间包含为系统定义的所有段和系统表。

为了将逻辑地址转换为线性地址，处理器执行以下操作：

1. 使用段选择子中的偏移量在GDT或LDT中定位段的段描述符d
将其读取到处理器中。（仅当新的段选择子被加载到段寄存器时才需要此步骤。）2. 检查段描述符以验证段的访问权限和范围，确保该段可访问且偏移量在段界限内。3. 将段描述符中的段基地址与偏移量相加，形成线性地址。

protection facilities. For example, it lets read-write protection be enforced on a page-by-page basis. The paging mechanism also provides two-level user-supervisor protection that can also be specified on a page-by-page basis.

3.3 PHYSICAL ADDRESS SPACE

In protected mode, the IA-32 architecture provides a normal physical address space of 4 GBytes (2^{32} bytes). This is the address space that the processor can address on its address bus. This address space is flat (unsegmented), with addresses ranging continuously from 0 to FFFFFFFFH. This physical address space can be mapped to read-write memory, read-only memory, and memory mapped I/O. The memory mapping facilities described in this chapter can be used to divide this physical memory up into segments and/or pages.

Starting with the Pentium Pro processor, the IA-32 architecture also supports an extension of the physical address space to 2^{36} bytes (64 GBytes); with a maximum physical address of FFFFFFFFFH. This extension is invoked in either of two ways:

- Using the physical address extension (PAE) flag, located in bit 5 of control register CR4.
- Using the 36-bit page size extension (PSE-36) feature (introduced in the Pentium III processors).

Physical address support has since been extended beyond 36 bits. See Chapter 4, “Paging” for more information about 36-bit physical addressing.

3.3.1 Intel® 64 Processors and Physical Address Space

On processors that support Intel 64 architecture (CPUID.80000001:EDX[29] = 1), the size of the physical address range is implementation-specific and indicated by CPUID.80000008H:EAX[bits 7-0].

For the format of information returned in EAX, see “CPUID—CPU Identification” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*. See also: Chapter 4, “Paging.”

3.4 LOGICAL AND LINEAR ADDRESSES

At the system-architecture level in protected mode, the processor uses two stages of address translation to arrive at a physical address: logical-address translation and linear address space paging.

Even with the minimum use of segments, every byte in the processor’s address space is accessed with a logical address. A logical address consists of a 16-bit segment selector and a 32-bit offset (see Figure 3-5). The segment selector identifies the segment the byte is located in and the offset specifies the location of the byte in the segment relative to the base address of the segment.

The processor translates every logical address into a linear address. A linear address is a 32-bit address in the processor’s linear address space. Like the physical address space, the linear address space is a flat (unsegmented), 2^{32}-byte address space, with addresses ranging from 0 to FFFFFFFFH. The linear address space contains all the segments and system tables defined for a system.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to locate the segment descriptor for the segment in the GDT or LDT and reads it into the processor. (This step is needed only when a new segment selector is loaded into a segment register.)
2. Examines the segment descriptor to check the access rights and range of the segment to insure that the segment is accessible and that the offset is within the limits of the segment.
3. Adds the base address of the segment from the segment descriptor to the offset to form a linear address.

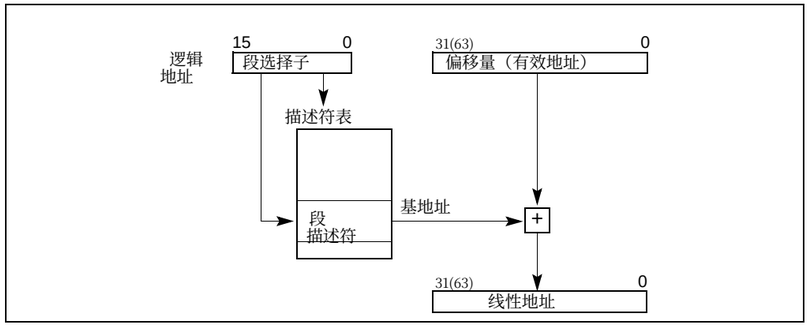


图3-5. 逻辑地址到线性地址的转换

若未启用分页机制，处理器会将线性地址直接映射为物理地址（即线性地址通过处理器的地址总线输出）。若线性地址空间启用了分页，则通过第二级地址转换将线性地址转换为物理地址。

另请参阅：第4章“分页”。

3.4.1 IA-32e模式中的逻辑地址转换

在IA-32e模式下，Intel 64处理器使用上述步骤将逻辑地址转换为线性地址。在64位模式下，段的偏移量和基地址为64位而非32位。线性地址格式也为64位宽，并需满足规范形式要求。

每个代码段描述符都提供了一个L位。该位允许代码段按代码段执行64位代码或传统32位代码。

3.4.2 段选择子

段选择子是一个16位的段标识符（参见图3-6）。它并不直接指向段，而是指向定义该段的段描述符。段选择子包含以下内容：

索引　　（位3至15）——从GDT或LDT的8192个描述符中选择一个。处理器将索引值乘以8（段描述符的字节数），并将结果与GDT或LDT的基地址（分别来自GDTR或LDTR寄存器）相加。

TI（表指示符）标志（位2）——Sp

指定要使用的描述符表：清除此标志选择GDT；设置此标志选择当前LDT。

图 3-6. 段选择子

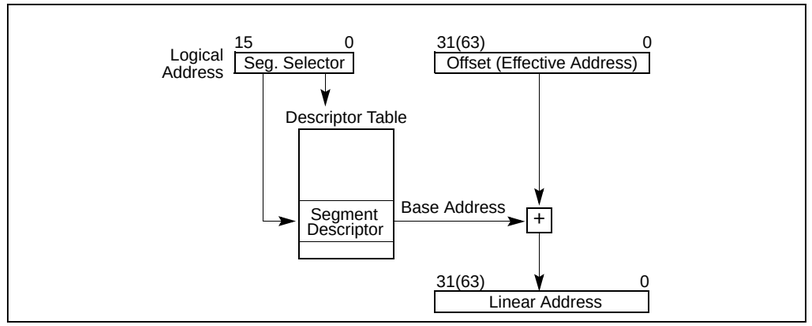


Figure 3-5. Logical Address to Linear Address Translation

If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor’s address bus). If the linear address space is paged, a second level of address translation is used to translate the linear address into a physical address.

See also: Chapter 4, “Paging.”

3.4.1 Logical Address Translation in IA-32e Mode

In IA-32e mode, an Intel 64 processor uses the steps described above to translate a logical address to a linear address. In 64-bit mode, the offset and base address of the segment are 64-bits instead of 32 bits. The linear address format is also 64 bits wide and is subject to the canonical form requirement.

Each code segment descriptor provides an L bit. This bit allows a code segment to execute 64-bit code or legacy 32-bit code by code segment.

3.4.2 Segment Selectors

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

Index　　(Bits 3 through 15) — Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).

TI (table indicator) flag

(Bit 2) — Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

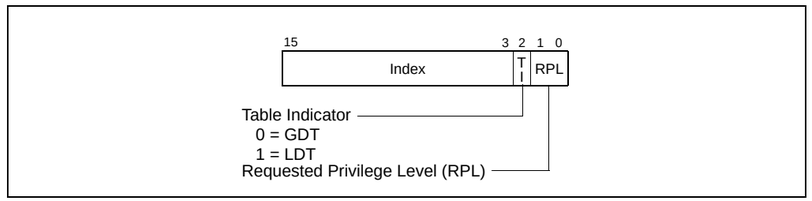


Figure 3-6. Segment Selector

请求特权级（RPL）（位0和位1）— 规范

指定选择器的特权级。特权级范围从0到3，其中0为最高特权级。有关RPL与正在执行的程序（或任务）的CPL以及段选择子所指向的描述符的描述符特权级（DPL）之间关系的说明，请参阅第5.5节“特权级”。

GDT的第一个条目不被处理器使用。指向GDT此条目的段选择子（即索引为0且TI标志设置为0的段选择子）被用作“空段选择子”。当段寄存器（CS或SS寄存器除外）加载空选择子时，处理器不会生成异常。然而，当持有空选择子的段寄存器用于访问内存时，处理器会生成异常。空选择子可用于初始化未使用的段寄存器。用空段选择子加载CS或SS寄存器会导致生成通用保护异常（#GP）。

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分配或修改，而非由应用程序修改。

3.4.3 段寄存器

为减少地址转换时间和编码复杂度，处理器提供了用于保存最多6个段选择子的寄存器（见图3-7）。每个段寄存器支持特定类型的内存引用（代码、堆栈或数据）。virtually任何程序执行要发生，至少需要将代码段（CS）、数据段（DS）和堆栈段（SS）寄存器加载有效的段选择子。处理器还提供三个额外的数据段寄存器（ES、FS和GS），可用于使当前执行的程序（或任务）能够访问额外的数据段。

程序要访问某个段，该段的段选择子必须已加载到某个段寄存器中。因此，虽然系统可以定义数千个段，但只有6个能够立即使用。其他段可以通过在程序执行期间将其段选择子加载到这些寄存器中来变为可用。

图3-7 段寄存器

每个段寄存器都有一个“可见部分”和一个“隐藏部分”。（隐藏部分有时被称为“描述符缓存”或“影子寄存器”。）当段选择子被加载到段寄存器的可见部分时，处理器还会将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段寄存器的隐藏部分。缓存在段寄存器（可见和隐藏）中的信息使处理器能够转换地址，而无需额外的总线周期从段描述符中读取基地址和界限。在多个处理器访问同一描述符表的系统中，软件有责任在修改描述符表时重新加载段寄存器。如果不这样做，则可能在内存驻留版本的段描述符被修改后，仍使用缓存在段寄存器中的旧段描述符。

提供了两种加载指令用于加载段寄存器：

1. 直接加载指令，例如MOV指令、POP指令、LDS指令、LES指令、LSS指令、LGS指令和LFS指令。这些指令显式引用段寄存器。

Requested Privilege Level (RPL)

(Bits 0 and 1) — Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level. See Section 5.5, “Privilege Levels”, for a description of the relationship of the RPL to the CPL of the executing program (or task) and the descriptor privilege level (DPL) of the descriptor the segment selector points to.

The first entry of the GDT is not used by the processor. A segment selector that points to this entry of the GDT (that is, a segment selector with an index of 0 and the TI flag set to 0) is used as a “null segment selector.” The processor does not generate an exception when a segment register (other than the CS or SS registers) is loaded with a null selector. It does, however, generate an exception when a segment register holding a null selector is used to access memory. A null selector can be used to initialize unused segment registers. Loading the CS or SS register with a null segment selector causes a general-protection exception (#GP) to be generated.

Segment selectors are visible to application programs as part of a pointer variable, but the values of selectors are usually assigned or modified by link editors or linking loaders, not application programs.

3.4.3 Segment Registers

To reduce address translation time and coding complexity, the processor provides registers for holding up to 6 segment selectors (see Figure 3-7). Each of these segment registers support a specific kind of memory reference (code, stack, or data). For virtually any kind of program execution to take place, at least the code-segment (CS), data-segment (DS), and stack-segment (SS) registers must be loaded with valid segment selectors. The processor also provides three additional data-segment registers (ES, FS, and GS), which can be used to make additional data segments available to the currently executing program (or task).

For a program to access a segment, the segment selector for the segment must have been loaded in one of the segment registers. So, although a system can define thousands of segments, only 6 can be available for immediate use. Other segments can be made available by loading their segment selectors into these registers during program execution.

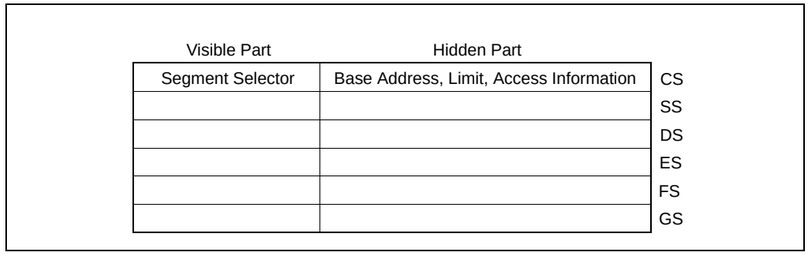


Figure 3-7. Segment Registers

Every segment register has a “visible” part and a “hidden” part. (The hidden part is sometimes referred to as a “descriptor cache” or a “shadow register.”) When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor pointed to by the segment selector. The information cached in the segment register (visible and hidden) allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor. In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified. If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified.

Two kinds of load instructions are provided for loading the segment registers:

1. Direct load instructions such as the MOV, POP, LDS, LES, LSS, LGS, and LFS instructions. These instructions explicitly reference the segment registers.

2. 隐式加载指令，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。这些指令在执行过程中会附带改变CS寄存器（有时还包括其他段寄存器）的内容。

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。

3.4.4 IA-32e模式中的段加载指令

由于ES、DS和SS段寄存器在64位模式下不被使用，它们在段描述符寄存器中的字段（基址、界限和属性）将被忽略。某些形式的段加载指令也无效（例如LDS指令、POP ES）。引用ES、DS或SS段的地址计算会被视为段基址为零来处理。

处理器会检查所有线性地址引用是否均为规范形式，而非执行界限检查。模式切换不会改变段寄存器或相关描述符寄存器的内容。这些寄存器在64位模式执行期间也不会改变，除非执行了显式的段加载操作。

为了为应用程序设置兼容模式，段加载指令（MOV到段寄存器、POP段寄存器）在64位模式下正常工作。从系统描述符表（GDT或LDT）中读取一个条目，并加载到段描述符寄存器的隐藏部分。描述符寄存器的基地址、界限和属性字段都会被加载。但是，数据和堆栈段选择器及描述符寄存器的内容将被忽略。

当在64位模式下使用FS和GS段超越时，它们各自的基地址将用于线性地址计算：(FS或GS).base + 索引 + 位移量。随后FS.base和GS.base会扩展至实现所支持的完整线性地址大小。最终的有效地址计算可跨越正负地址环绕；生成的线性地址必须是规范地址。

在64位模式下，使用FS段和GS段超越的内存访问不会检查运行时界限，也不会进行属性检查。正常的段加载（MOV到段寄存器和POP段寄存器）到FS和GS时，会在段描述符寄存器的隐藏部分加载标准的32位基址值。高于标准32位的基地址位会被清零，以确保使用少于64位的实现保持一致性。

FS.base 和 GS.base 的隐藏描述符寄存器字段被物理映射到模型特定寄存器，以便加载64位实现支持的所有地址位。具有当前特权级 = 0 （特权软件）的软件可以使用WRMSR指令将所有支持的线性地址位加载到 FS.base 或 GS.base 中。写入64位 FS.base 和 GS.base 寄存器的地址必须采用规范形式。试图向这些寄存器写入非规范地址的 WRMSR 指令将导致#GP故障。

在兼容模式下，无论加载到隐藏描述符寄存器基址字段的高32位线性地址位的值如何，FS 和 GS 覆盖操作都按照32位模式行为定义执行。兼容模式在计算有效地址时会忽略高32位。

新增的64位模式指令 SWAPGS 可用于加载 GS 基址。SWAPGS 将来自 IA32_KernelGSbase 模型特定寄存器的内核数据结构指针与 GS 基址寄存器进行交换。随后内核可在常规内存引用中使用 GS 前缀来访问内核数据结构。尝试向 IA32_KernelGSBase 模型特定寄存器写入非规范值（使用 WRMSR）将导致#GP故障。

3.4.5 段描述符

段描述符是GDT或LDT中的一种数据结构，它为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、加载器或操作系统/执行体创建，而非应用程序。图3-8展示了所有类型段描述符的通用描述符格式。

2. Implied load instructions such as the far pointer versions of the CALL, JMP, and RET instructions, the SYSENTER and SYSEXIT instructions, and the IRET, INT*n*, INTO and INT3 instructions. These instructions change the contents of the CS register (and sometimes other segment registers) as an incidental part of their operation.

The MOV instruction can also be used to store visible part of a segment register in a general-purpose register.

3.4.4 Segment Loading Instructions in IA-32e Mode

Because ES, DS, and SS segment registers are not used in 64-bit mode, their fields (base, limit, and attribute) in segment descriptor registers are ignored. Some forms of segment load instructions are also invalid (for example, LDS, POP ES). Address calculations that reference the ES, DS, or SS segments are treated as if the segment base is zero.

The processor checks that all linear-address references are in canonical form instead of performing limit checks. Mode switching does not change the contents of the segment registers or the associated descriptor registers. These registers are also not changed during 64-bit mode execution, unless explicit segment loads are performed.

In order to set up compatibility mode for an application, segment-load instructions (MOV to Sreg, POP Sreg) work normally in 64-bit mode. An entry is read from the system descriptor table (GDT or LDT) and is loaded in the hidden portion of the segment descriptor register. The descriptor-register base, limit, and attribute fields are all loaded. However, the contents of the data and stack segment selector and the descriptor registers are ignored.

When FS and GS segment overrides are used in 64-bit mode, their respective base addresses are used in the linear address calculation: (FS or GS).base + index + displacement. FS.base and GS.base are then expanded to the full linear-address size supported by the implementation. The resulting effective address calculation can wrap across positive and negative addresses; the resulting linear address must be canonical.

In 64-bit mode, memory accesses using FS-segment and GS-segment overrides are not checked for a runtime limit nor subjected to attribute-checking. Normal segment loads (MOV to Sreg and POP Sreg) into FS and GS load a standard 32-bit base value in the hidden portion of the segment descriptor register. The base address bits above the standard 32 bits are cleared to 0 to allow consistency for implementations that use less than 64 bits.

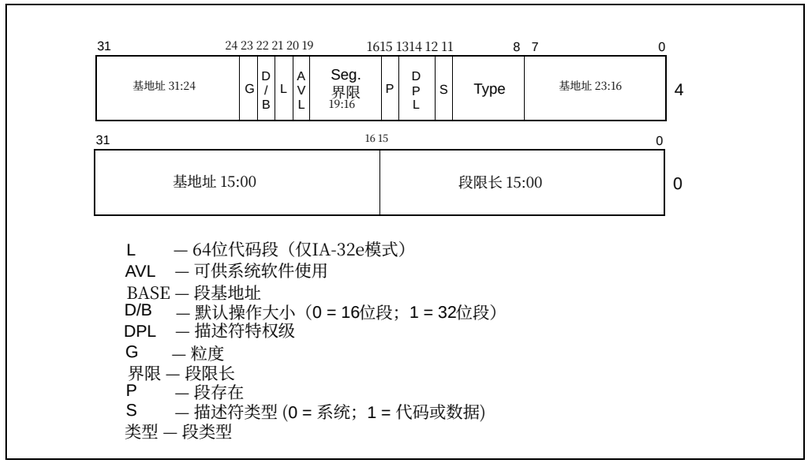
The hidden descriptor register fields for FS.base and GS.base are physically mapped to MSRs in order to load all address bits supported by a 64-bit implementation. Software with CPL = 0 (privileged software) can load all supported linear-address bits into FS.base or GS.base using WRMSR. Addresses written into the 64-bit FS.base and GS.base registers must be in canonical form. A WRMSR instruction that attempts to write a non-canonical address to those registers causes a #GP fault.

When in compatibility mode, FS and GS overrides operate as defined by 32-bit mode behavior regardless of the value loaded into the upper 32 linear-address bits of the hidden descriptor register base field. Compatibility mode ignores the upper 32 bits when calculating an effective address.

A new 64-bit mode instruction, SWAPGS, can be used to load GS base. SWAPGS exchanges the kernel data structure pointer from the IA32_KernelGSbase MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access the kernel data structures. An attempt to write a non-canonical value (using WRMSR) to the IA32_KernelGSBase MSR causes a #GP fault.

3.4.5 Segment Descriptors

A segment descriptor is a data structure in a GDT or LDT that provides the processor with the size and location of a segment, as well as access control and status information. Segment descriptors are typically created by compilers, linkers, loaders, or the operating system or executive, but not application programs. Figure 3-8 illustrates the general descriptor format for all types of segment descriptors.



L — 64位代码段（仅IA-32e模式）
AVL — 可供系统软件使用
BASE — 段基地址
D/B — 默认操作大小（0 = 16位段；1 = 32位段）
DPL — 描述符特权级
G — 粒度
界限 — 段限长
P — 段存在
S — 描述符类型 (0 = 系统；1 = 代码或数据)
类型 — 段类型

图3-8 段描述符

段描述符中的标志位和字段如下：

段限界字段说明

用于指定段的大小。处理器将两个段限界字段组合形成一个20位的值。根据G（粒度）标志的设置，处理器以两种方式之一解释段限界：

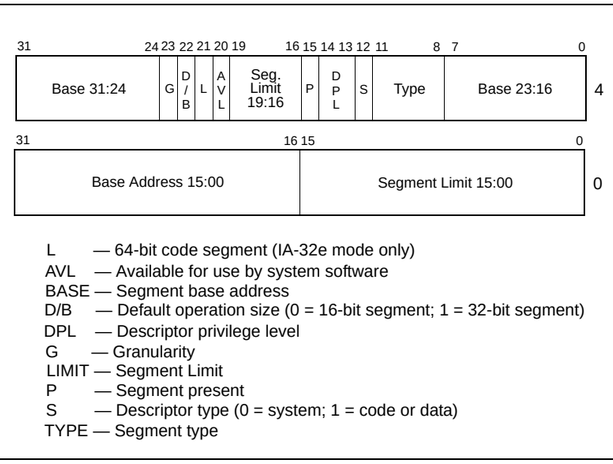
- 如果粒度标志未设置，段大小范围可从1字节到1MB，以字节为单位递增。
- 如果粒度标志已设置，段大小范围可从4KB到4GB，以4KB为单位递增。

处理器以两种不同方式使用段限界，具体取决于该段是向上扩展段还是向下扩展段。有关段类型的更多信息，请参阅第3.4.5.1节“代码段与数据段描述符类型”。对于向上扩展段，逻辑地址中的偏移量范围可从0至段限界。超过段限界的偏移量将引发通用保护异常（#GP，适用于除SS段外的所有段）或堆栈故障异常（#SS适用于SS段）。对于向下扩展段，段限界发挥反向作用：偏移量范围可从段限界加1至FFFFFFFFH或FFFFH，具体取决于B标志的设置。小于或等于段限界的偏移量会引发通用保护异常或堆栈故障异常。减小向下扩展段的段限界字段值会在段地址空间的底部（而非顶部）分配新内存。IA-32架构的堆栈始终向下增长，这使得该机制非常适合可扩展堆栈。

基地址字段定义

段内字节0在4GB线性地址空间中的位置。处理器将三个基地址字段组合形成单个32位值。段基地址应对齐到16字节边界。虽然不要求必须16字节对齐，但这种对齐方式允许程序通过将代码和数据对齐到16字节边界来最大化性能。

类型字段　指示段或门类型，并指定可对段进行的访问类型及增长方向。该字段的解释取决于描述符类型标志指定的是应用程序（代码或数据）描述符还是系统描述符。对于代码、数据和系统描述符，类型字段的编码方式各不相同（见图5-1）。关于如何使用此字段指定代码段和数据段类型的描述，请参阅第3.4.5.1节“代码段与数据段描述符类型”。



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

The flags and fields in a segment descriptor are as follows:

Segment limit field
Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

- If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
- If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

The processor uses the segment limit in two different ways, depending on whether the segment is an expand-up or an expand-down segment. See Section 3.4.5.1, “Code- and Data-Segment Descriptor Types”, for more information about segment types. For expand-up segments, the offset in a logical address can range from 0 to the segment limit. Offsets greater than the segment limit generate general-protection exceptions (#GP, for all segment other than SS) or stack-fault exceptions (#SS for the SS segment). For expand-down segments, the segment limit has the reverse function; the offset can range from the segment limit plus 1 to FFFFFFFFH or FFFFH, depending on the setting of the B flag. Offsets less than or equal to the segment limit generate general-protection exceptions or stack-fault exceptions. Decreasing the value in the segment limit field for an expand-down segment allocates new memory at the bottom of the segment's address space, rather than at the top. IA-32 architecture stacks always grow downwards, making this mechanism convenient for expandable stacks.

Base address fields
Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although 16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

Type field　Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors (see Figure 5-1). See Section 3.4.5.1, “Code- and Data-Segment Descriptor Types”, for a description of how this field is used to specify code and data-segment types.

S（描述符类型）标志 指定

段描述符是用于系统段（S标志清零）还是代码段或数据段（S标志置位）。

DPL（描述符特权级）字段 指定段的特权级

。特权级范围从0到3，其中0为最高特权级。DPL用于控制对段的访问。有关DPL与执行代码段的CPL及段选择子的RPL之间关系的描述，请参见第5.5节“特权级”。

P（段存在）标志 指示

该段是否存在于内存中（置位）或不存在（清除）。如果该标志位被清除，当指向该段描述符的段选择子被加载到段寄存器时，处理器会产生一个段不存在异常（#NP）。内存管理软件可以利用此标志来控制哪些段在特定时刻实际加载到物理内存中。这为管理虚拟内存提供了除分页之外的额外控制手段。

图3-9展示了当段存在标志位被清除时段描述符的格式。当该标志位被清除时，操作系统或执行体可以自由使用标记为“可用”的位置来存储自身数据，例如关于缺失段位置的信息。

D/B（默认操作大小/默认堆栈指针大小和/或上界）标志 根据段描述符类型执行不同功能

对于可执行代码段、向下扩展数据段或堆栈段，该标志的功能各不相同。（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，则应设置为0。）

- 可执行代码段。该标志称为D标志，它指示段内指令引用的有效地址和操作数的默认长度。如果该标志被设置，则假定为32位地址和32位或8位操作数；如果该标志被清除，则假定为16位地址和16位或8位操作数。指令前缀66H可用于选择非默认的操作数大小，前缀67H可用于选择非默认的地址大小。

- 堆栈段（由SS寄存器指向的数据段）。该标志称为B（大）标志，它指定用于隐式堆栈操作（如压栈、出栈和调用）的堆栈指针大小。如果该标志被设置，则使用32位堆栈指针，存储在32位ESP寄存器中；如果该标志被清除，则使用16位堆栈指针，存储在16位SP寄存器中。如果堆栈段被设置为向下扩展数据段（在下一段中描述），B标志还指定堆栈段的上界。

- 向下扩展数据段。该标志称为B标志，它指定了段的上界。如果设置了该标志，上界为FFFFFFFFH（4 GB）；如果清除了该标志，上界为FFFFH（64 KB）。

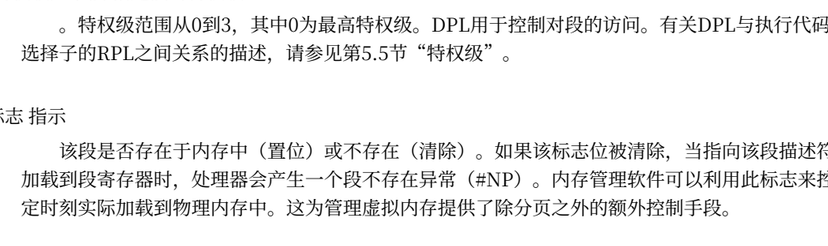


图3-9. 段存在标志位清除时的段描述符

G（粒度）标志确定

定义了段限界字段的缩放比例。当粒度标志清零时，段限长以字节单位解释；当该标志置位时，段限长以4KB单位解释。（此标志不影响基地址的粒度；基地址始终以字节粒度表示。）当粒度标志置位时，检查偏移量与段限长的匹配性时，偏移量的12个最低有效位不会被检测。

S (descriptor type) flag

Specifies whether the segment descriptor is for a system segment (S flag is clear) or a code or data segment (S flag is set).

DPL (descriptor privilege level) field

Specifies the privilege level of the segment. The privilege level can range from 0 to 3, with 0 being the most privileged level. The DPL is used to control access to the segment. See Section 5.5, “Privilege Levels”, for a description of the relationship of the DPL to the CPL of the executing code segment and the RPL of a segment selector.

P (segment-present) flag

Indicates whether the segment is present in memory (set) or not present (clear). If this flag is clear, the processor generates a segment-not-present exception (#NP) when a segment selector that points to the segment descriptor is loaded into a segment register. Memory management software can use this flag to control which segments are actually loaded into physical memory at a given time. It offers a control in addition to paging for managing virtual memory.

Figure 3-9 shows the format of a segment descriptor when the segment-present flag is clear. When this flag is clear, the operating system or executive is free to use the locations marked “Available” to store its own data, such as information regarding the whereabouts of the missing segment.

D/B (default operation size/default stack pointer size and/or upper bound) flag

Performs different functions depending on whether the segment descriptor is an executable code segment, an expand-down data segment, or a stack segment. (This flag should always be set to 1 for 32-bit code and data segments and to 0 for 16-bit code and data segments.)

- **Executable code segment.** The flag is called the D flag and it indicates the default length for effective addresses and operands referenced by instructions in the segment. If the flag is set, 32-bit addresses and 32-bit or 8-bit operands are assumed; if it is clear, 16-bit addresses and 16-bit or 8-bit operands are assumed.
  The instruction prefix 66H can be used to select an operand size other than the default, and the prefix 67H can be used select an address size other than the default.

- **Stack segment (data segment pointed to by the SS register).** The flag is called the B (big) flag and it specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls). If the flag is set, a 32-bit stack pointer is used, which is stored in the 32-bit ESP register; if the flag is clear, a 16-bit stack pointer is used, which is stored in the 16-bit SP register. If the stack segment is set up to be an expand-down data segment (described in the next paragraph), the B flag also specifies the upper bound of the stack segment.

- **Expand-down data segment.** The flag is called the B flag and it specifies the upper bound of the segment. If the flag is set, the upper bound is FFFFFFFFH (4 GBytes); if the flag is clear, the upper bound is FFFFH (64 KBytes).

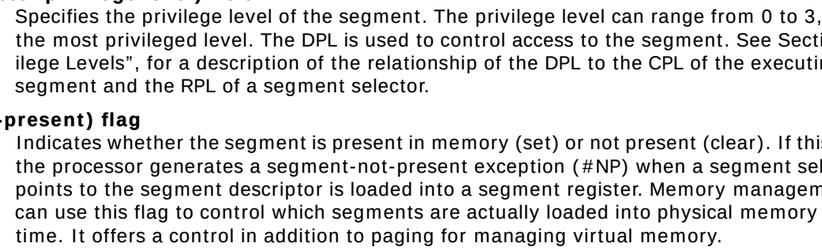


Figure 3-9. Segment Descriptor When Segment-Present Flag Is Clear

G (granularity) flag

Determines the scaling of the segment limit field. When the granularity flag is clear, the segment limit is interpreted in byte units; when flag is set, the segment limit is interpreted in 4-KByte units. (This flag does not affect the granularity of the base address; it is always byte granular.) When the granularity flag is set, the twelve least significant bits of an offset are not tested when checking the

例如，当粒度标志置位时，限长为0将使得从0到4095的偏移量均有效。

L（64位代码段）标志 在IA-32e模式下，

段描述符第二个双字的第21位指示代码段是否包含原生64位代码。值为1表示此代码段中的指令在64位模式下执行。值为0表示此代码段中的指令在兼容模式下执行。如果设置了L位，则必须清除D位。当不在IA-32e模式下或对于非代码段时，第21位被保留且应始终设置为0。

可用与保留位 段描述符第二个双字的第20位

可供系统软件使用。

3.4.5.1 代码段与数据段描述符类型

当段描述符中的S（描述符类型）标志被设置时，该描述符用于代码段或数据段。此时类型字段的最高有效位（段描述符第二个双字的第11位）决定了该描述符是用于数据段（清零）还是代码段（置位）。

对于数据段，类型字段的三个低有效位（第8、9和10位）被解释为访问位 (A)、写使能 (W) 和扩展方向 (E)。有关代码段和数据段类型字段位编码的说明，请参见表3-1。根据写使能位的设置，数据段可以是只读段或读/写段。

表3-1. 代码段与数据段类型

类型字段					描述符 Type	描述
十进制	11	10 E	9 W	8 A		
0	0	0	0	0	Data	只读
1	0	0	0	1	Data	只读，已访问
2	0	0	1	0	Data	读/写
3	0	0	1	1	Data	读/写 ，已访问
4	0	1	0	0	Data	只读, 向下扩展
5	0	1	0	1	Data	只读, 向下扩展, 已访问
6	0	1	1	0	Data	读/写, 向下扩展
7	0	1	1	1	Data	读/写, 向下扩展, 已访问
		C	R	A		
8	1	0	0	0	Code	仅执行
9	1	0	0	1	Code	仅执行，已访问
10	1	0	1	0	Code	执行/读取
11	1	0	1	1	Code	执行/读取 ，已访问
12	1	1	0	0	Code	仅执行，符合性
13	1	1	0	1	Code	仅执行，符合性，已访问
14	1	1	1	0	Code	执行/读取，符合性
15	1	1	1	1	Code	执行/读取，符合性，已访问

堆栈段是必须为读/写段的数据段。使用不可写数据段的段选择子加载SS寄存器会引发通用保护异常（#GP）。如果需要动态更改堆栈段的大小，该堆栈段可以是向下扩展数据段（扩展方向标志已设置）。此时，动态更改段限长会将堆栈空间添加到

offset against the segment limit. For example, when the granularity flag is set, a limit of 0 results in valid offsets from 0 to 4095.

L (64-bit code segment) flag

In IA-32e mode, bit 21 of the second doubleword of the segment descriptor indicates whether a code segment contains native 64-bit code. A value of 1 indicates instructions in this code segment are executed in 64-bit mode. A value of 0 indicates the instructions in this code segment are executed in compatibility mode. If L-bit is set, then D-bit must be cleared. When not in IA-32e mode or for non-code segments, bit 21 is reserved and should always be set to 0.

Available and reserved bits

Bit 20 of the second doubleword of the segment descriptor is available for use by system software.

3.4.5.1 Code- and Data-Segment Descriptor Types

When the S (descriptor type) flag in a segment descriptor is set, the descriptor is for either a code or a data segment. The highest order bit of the type field (bit 11 of the second double word of the segment descriptor) then determines whether the descriptor is for a data segment (clear) or a code segment (set).

For data segments, the three low-order bits of the type field (bits 8, 9, and 10) are interpreted as accessed (A), write-enable (W), and expansion-direction (E). See Table 3-1 for a description of the encoding of the bits in the type field for code and data segments. Data segments can be read-only or read/write segments, depending on the setting of the write-enable bit.

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

Stack segments are data segments which must be read/write segments. Loading the SS register with a segment selector for a nonwritable data segment generates a general-protection exception (#GP). If the size of a stack segment needs to be changed dynamically, the stack segment can be an expand-down data segment (expansion-direction flag set). Here, dynamically changing the segment limit causes stack space to be added to the bottom of

堆栈底部。如果堆栈段的大小需要保持静态，则该堆栈段可以是向上扩展或向下扩展类型。

访问位指示自上次操作系统或执行体清除该位以来，该段是否已被访问。当处理器将段的段选择子加载到段寄存器时，假设包含段描述符的内存类型支持处理器写入，便会设置该位。该位将保持设置状态，直到被显式清除。此位既可用于虚拟内存管理，也可用于调试。

对于代码段，类型字段的低三位被解释为已访问 (A)、读使能 (R) 和符合性 (C)。代码段可以是仅执行或执行/读取，具体取决于读使能位的设置。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。在这种情况下，可以通过使用带有CS覆盖前缀的指令，或者通过将代码段的段选择子加载到数据段寄存器（DS、ES、FS或GS寄存器）中，从代码段读取数据。在保护模式下，代码段不可写入。

代码段可以是符合性或非一致类型。向更高特权级的符合性段转移执行时，允许在当前特权级继续执行。而向不同特权级的非一致段转移执行将导致通用保护异常（#GP），除非使用调用门或任务门（有关符合性和非一致代码段的更多信息，请参阅第5.8.1节“直接调用或跳转至代码段”）。不访问受保护设施的系统工具以及某些异常类型（例如除法错误或溢出）的处理程序，可被加载到符合性代码段中。需要防范低特权级程序和过程访问的实用程序应置于非一致代码段中。

NOTE

无论目标段是符合性还是非一致代码段，都无法通过调用或跳转将执行权转移至更低特权级（数值上更高的特权级）的代码段。尝试此类执行转移将触发通用保护异常。

所有数据段都是非一致的，这意味着它们不能被特权级较低的程序或过程（在数值上较高特权级执行的代码）访问。然而，与代码段不同，数据段可以被特权级较高的程序或过程（在数值上较低特权级执行的代码）访问，而无需使用特殊的访问门。

如果GDT或某个LDT中的段描述符被放置在ROM中，当软件或处理器尝试更新（写入）这些基于ROM的段描述符时，处理器可能会进入无限循环。为防止此问题，应为所有放置在ROM中的段描述符设置访问位。同时，移除那些试图修改位于ROM中的段描述符的操作系统或执行体代码。

3.5 系统描述符类型

当段描述符中的S（描述符类型）标志被清除时，该描述符类型为系统描述符。处理器可识别以下类型的系统描述符：

- 局部描述符表（LDT）段描述符
- 任务状态段（TSS）描述符。
- 调用门描述符。
- 中断门描述符。
- 陷阱门描述符。
- 任务门描述符。

r.

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）。门描述符本身即是“门”，它们持有指向代码段中过程入口点的指针（调用门、中断门和陷阱门），或者持有TSS的段选择子（任务门）。

the stack. If the size of a stack segment is intended to remain static, the stack segment may be either an expand-up or expand-down type.

The accessed bit indicates whether the segment has been accessed since the last time the operating-system or executive cleared the bit. The processor sets this bit whenever it loads a segment selector for the segment into a segment register, assuming that the type of memory that contains the segment descriptor supports processor writes. The bit remains set until explicitly cleared. This bit can be used both for virtual memory management and for debugging.

For code segments, the three low-order bits of the type field are interpreted as accessed (A), read enable (R), and conforming (C). Code segments can be execute-only or execute/read, depending on the setting of the read-enable bit. An execute/read segment might be used when constants or other static data have been placed with instruction code in a ROM. Here, data can be read from the code segment either by using an instruction with a CS override prefix or by loading a segment selector for the code segment in a data-segment register (the DS, ES, FS, or GS registers). In protected mode, code segments are not writable.

Code segments can be either conforming or nonconforming. A transfer of execution into a more-privileged conforming segment allows execution to continue at the current privilege level. A transfer into a nonconforming segment at a different privilege level results in a general-protection exception (#GP), unless a call gate or task gate is used (see Section 5.8.1, “Direct Calls or Jumps to Code Segments”, for more information on conforming and nonconforming code segments). System utilities that do not access protected facilities and handlers for some types of exceptions (such as, divide error or overflow) may be loaded in conforming code segments. Utilities that need to be protected from less privileged programs and procedures should be placed in nonconforming code segments.

NOTE

Execution cannot be transferred by a call or a jump to a less-privileged (numerically higher privilege level) code segment, regardless of whether the target segment is a conforming or nonconforming code segment. Attempting such an execution transfer will result in a general-protection exception.

All data segments are nonconforming, meaning that they cannot be accessed by less privileged programs or procedures (code executing at numerically high privilege levels). Unlike code segments, however, data segments can be accessed by more privileged programs or procedures (code executing at numerically lower privilege levels) without using a special access gate.

If the segment descriptors in the GDT or an LDT are placed in ROM, the processor can enter an indefinite loop if software or the processor attempts to update (write to) the ROM-based segment descriptors. To prevent this problem, set the accessed bits for all segment descriptors placed in a ROM. Also, remove operating-system or executive code that attempts to modify segment descriptors located in ROM.

3.5 SYSTEM DESCRIPTOR TYPES

When the S (descriptor type) flag in a segment descriptor is clear, the descriptor type is a system descriptor. The processor recognizes the following types of system descriptors:

- Local descriptor-table (LDT) segment descriptor.
- Task-state segment (TSS) descriptor.
- Call-gate descriptor.
- Interrupt-gate descriptor.
- Trap-gate descriptor.
- Task-gate descriptor.

These descriptor types fall into two categories: system-segment descriptors and gate descriptors. System-segment descriptors point to system segments (LDT and TSS segments). Gate descriptors are in themselves “gates,” which hold pointers to procedure entry points in code segments (call, interrupt, and trap gates) or which hold segment selectors for TSS’s (task gates).

表3-2展示了系统段描述符和门描述符的类型字段编码。请注意，IA-32e模式下的系统描述符为16字节而非8字节。

表 3-2. 系统段和门描述符类型

类型字段					描述	
十进制	11	10	9	8	32位模式	IA-32e模式
0	0	0	0	0	保留	16字节描述符的高8字节描述符
1	0	0	0	1	16位TSS（可用）	保留
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16位TSS（繁忙）	保留
4	0	1	0	0	16位调用门	保留
5	0	1	0	1	任务门	保留
6	0	1	1	0	16位中断门	保留
7	0	1	1	1	16位陷阱门	保留
8	1	0	0	0	保留	保留
9	1	0	0	1	32位任务状态段（可用）	64位任务状态段（可用）
10	1	0	1	0	保留	保留
11	1	0	1	1	32位任务状态段（忙碌）	64位任务状态段（忙碌）
12	1	1	0	0	32位调用门	64位调用门
13	1	1	0	1	保留	保留
14	1	1	1	0	32位中断门	64位中断门
15	1	1	1	1	32位陷阱门	64位陷阱门

另请参阅：第3.5.1节“段描述符表”和第7.2.2节“TSS描述符”（有关系统段描述符的更多信息）；请参阅第5.8.3节“调用门”、第6.11节“IDT描述符”和第7.2.5节“任务门描述符”（有关门描述符的更多信息）。

3.5.1 段描述符表

段描述符表是段描述符的数组（参见图3-10）。描述符表的长度可变，最多可包含8192（213）个8字节描述符。描述符表有两种类型：

- 全局描述符表 (GDT)
- 局部描述符表 (LDT)

Table 3-2 shows the encoding of the type field for system-segment descriptors and gate descriptors. Note that system descriptors in IA-32e mode are 16 bytes instead of 8 bytes.

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

See also: Section 3.5.1, “Segment Descriptor Tables”, and Section 7.2.2, “TSS Descriptor” (for more information on the system-segment descriptors); see Section 5.8.3, “Call Gates”, Section 6.11, “IDT Descriptors”, and Section 7.2.5, “Task-Gate Descriptor” (for more information on the gate descriptors).

3.5.1 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors (see Figure 3-10). A descriptor table is variable in length and can contain up to 8192 (2^{13}) 8-byte descriptors. There are two kinds of descriptor tables:

- The global descriptor table (GDT)
- The local descriptor tables (LDT)

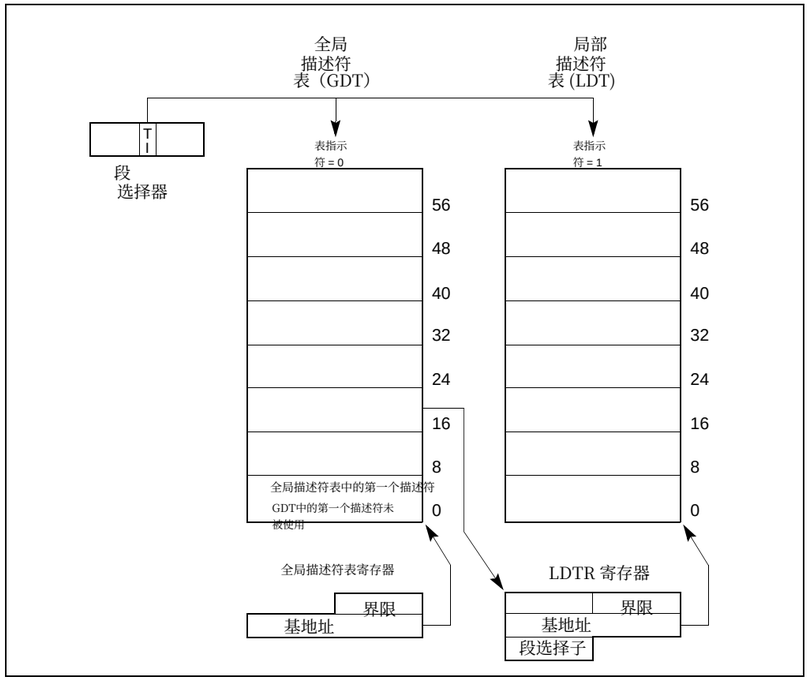


图 3-10. 全局和局部描述符表

每个系统必须定义一个GDT，可供系统中所有程序和任务使用。此外，还可以选择定义一个或多个LDT。例如，可以为每个正在运行的独立任务定义LDT，或者让部分或全部任务共享同一个LDT。

GDT本身并非一个段，而是线性地址空间中的一个数据结构。GDT的基线性地址和界限必须加载到GDTR寄存器中（参见第2.4节“内存管理寄存器”）。GDT的基地址应对齐到8字节边界以获得最佳处理器性能。GDT的界限值以字节为单位。与段类似，界限值会与基地址相加以得到最后一个有效字节的地址。界限值为0时恰好对应一个有效字节。由于段描述符始终为8字节长，GDT的界限值应始终为8的整数倍减一（即8N–1）。

GDT中的第一个描述符不被处理器使用。指向这个“空描述符”的段选择子被加载到数据段寄存器（DS、ES、FS或GS）时不会产生异常，但当尝试使用该描述符访问内存时，总是会引发通用保护异常（#GP）。通过用这个段选择子初始化段寄存器，可以确保对未使用段寄存器的意外引用必定会触发异常。

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 在 GDT 中都必须有独立的段选择子和段描述符。LDT 的段描述符可以位于 GDT 中的任意位置。有关 LDT 段描述符类型的信息，请参阅第 3.5 节“系统描述符类型”。

通过其段选择子来访问 LDT。为避免访问 LDT 时进行地址转换，LDT 的段选择子、基线性地址、界限和访问权限都存储在 LDTR 寄存器中（参见第 2.4 节“内存管理寄存器”）。

当存储GDTR寄存器（使用SGDT指令）时，一个48位“伪描述符”将被存入内存（参见图3-11顶部图示）。为避免用户模式（特权级3）下的对齐检查故障，该伪

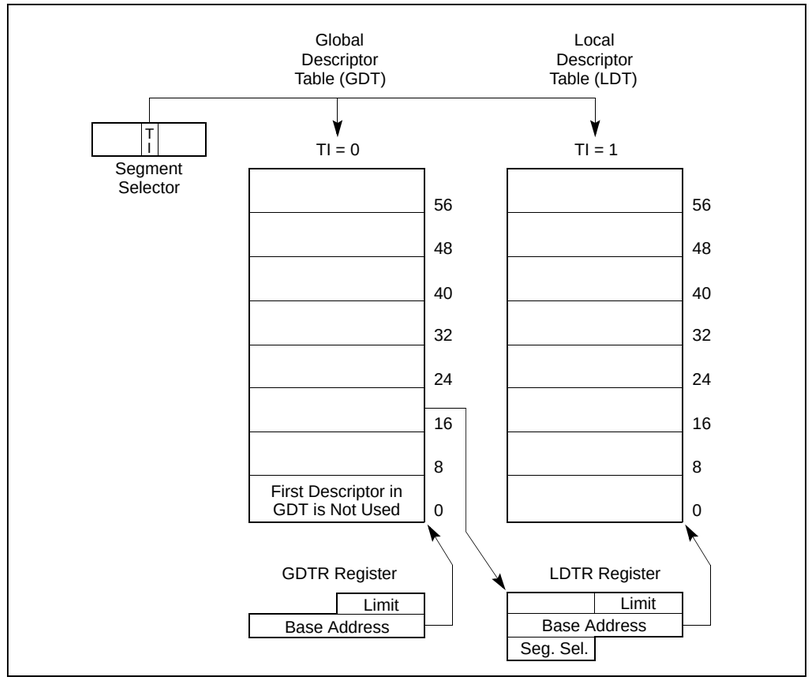


Figure 3-10. Global and Local Descriptor Tables

Each system must have one GDT defined, which may be used for all programs and tasks in the system. Optionally, one or more LDTs can be defined. For example, an LDT can be defined for each separate task being run, or some or all tasks can share the same LDT.

The GDT is not a segment itself; instead, it is a data structure in linear address space. The base linear address and limit of the GDT must be loaded into the GDTR register (see Section 2.4, “Memory-Management Registers”). The base addresses of the GDT should be aligned on an eight-byte boundary to yield the best processor performance. The limit value for the GDT is expressed in bytes. As with segments, the limit value is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly one valid byte. Because segment descriptors are always 8 bytes long, the GDT limit should always be one less than an integral multiple of eight (that is, 8N – 1).

The first descriptor in the GDT is not used by the processor. A segment selector to this “null descriptor” does not generate an exception when loaded into a data-segment register (DS, ES, FS, or GS), but it always generates a general-protection exception (#GP) when an attempt is made to access memory using the descriptor. By initializing the segment registers with this segment selector, accidental reference to unused segment registers can be guaranteed to generate an exception.

The LDT is located in a system segment of the LDT type. The GDT must contain a segment descriptor for the LDT segment. If the system supports multiple LDTs, each must have a separate segment selector and segment descriptor in the GDT. The segment descriptor for an LDT can be located anywhere in the GDT. See Section 3.5, “System Descriptor Types”, information on the LDT segment-descriptor type.

An LDT is accessed with its segment selector. To eliminate address translations when accessing the LDT, the segment selector, base linear address, limit, and access rights of the LDT are stored in the LDTR register (see Section 2.4, “Memory-Management Registers”).

When the GDTR register is stored (using the SGDT instruction), a 48-bit “pseudo-descriptor” is stored in memory (see top diagram in Figure 3-11). To avoid alignment check faults in user mode (privilege level 3), the pseudo-

描述符应位于奇数字地址（即地址MOD 4等于2）。这将使处理器存储一个对齐字，后跟一个对齐双字。用户模式程序通常不存储伪描述符，但通过以这种方式对齐伪描述符，可以避免产生对齐检查故障的可能性。在使用SIDT指令存储IDTR寄存器时，应采用相同的对齐方式。当存储LDTR或任务寄存器时（分别使用SLDT或STR指令），伪描述符应位于双字地址（即地址MOD 4等于0）。

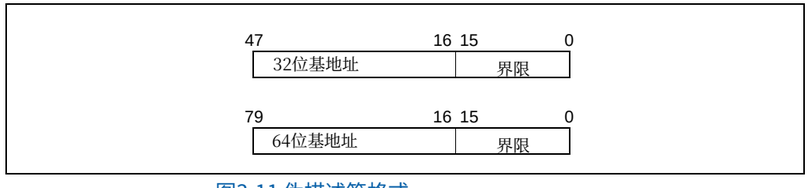


图3-11 伪描述符格式

3.5.2 IA-32e模式中的段描述符表

在IA-32e模式下，段描述符表最多可包含8192（213）个8字节描述符。段描述符表中的条目可以是8字节。系统描述符扩展为16字节（占用两个条目的空间）。

GDTR和LDTR寄存器扩展为可容纳64位基地址。相应的伪描述符为80位。（参见图3-11底部图示）。

以下系统描述符扩展为16字节：

— 调用门描述符（参见第5.8.3.1节“IA-32e模式调用门”）— IDT门描述符（参见第6.14.1节“64位模式IDT”）— LDT和TSS描述符（参见第7.2.3节“64位模式下的TSS描述符”）。

descriptor should be located at an odd word address (that is, address MOD 4 is equal to 2). This causes the processor to store an aligned word, followed by an aligned doubleword. User-mode programs normally do not store pseudo-descriptors, but the possibility of generating an alignment check fault can be avoided by aligning pseudo-descriptors in this way. The same alignment should be used when storing the IDTR register using the SIDT instruction. When storing the LDTR or task register (using the SLDT or STR instruction, respectively), the pseudo-descriptor should be located at a doubleword address (that is, address MOD 4 is equal to 0).

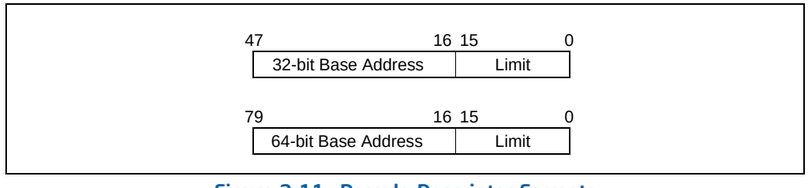


Figure 3-11. Pseudo-Descriptor Formats

3.5.2 Segment Descriptor Tables in IA-32e Mode

In IA-32e mode, a segment descriptor table can contain up to 8192 (2^{13}) 8-byte descriptors. An entry in the segment descriptor table can be 8 bytes. System descriptors are expanded to 16 bytes (occupying the space of two entries).

GDTR and LDTR registers are expanded to hold 64-bit base address. The corresponding pseudo-descriptor is 80 bits. (see the bottom diagram in Figure 3-11).

The following system descriptors expand to 16 bytes:

— Call gate descriptors (see Section 5.8.3.1, “IA-32e Mode Call Gates”)

— IDT gate descriptors (see Section 6.14.1, “64-Bit Mode IDT”)

— LDT and TSS descriptors (see Section 7.2.3, “TSS Descriptor in 64-bit mode”).