IA-32 架构软件开发人员手册

第 3 卷：系统编程指南

（中文版-部分）

目　　录

第 1 章 导　　读

《IA-32 Intel®架构软件开发人员手册 第 3 卷：系统编程指南》（订单号 245472）是描述 Intel 的 IA-32 处理器架构和开发环境的手册之一，其它两卷是：

- 《IA-32 Intel®架构软件开发人员手册 第 1 卷：基本架构》（订单号 245470）
- 《IA-32 Intel®架构软件开发人员手册 第 2 卷:指令集参考》(订单号 245471）

《IA-32 Intel®架构软件开发人员手册 第 1 卷：基本架构》（后文简称“《第 1 卷：基本架构》”）描述 Intel 的 IA-32 处理器的基本架构和编程环境。《IA-32 Intel®架构软件开发人员手册 第 2 卷：指令集参考》（后文简称“《第 2 卷：指令集参考》”）描述处理器的指令集和操作码结构。这两本手册主要供在现有操作系统之下写应用程序的开发人员参考。《IA-32 Intel®架构软件开发人员手册 第 3 卷：系统编程指南》（后文简称“《第 3 卷：系统编程指南》”）描述 IA-32 处理器的操作系统支撑环境，包括内存管理、保护、任务管理、中断和异常处理、以及系统管理等，同时也提供 IA-32 处理器的兼容信息。这卷手册主要供操作系统和 BIOS 开发人员参考。

1.1. 本手册涵盖的 IA-32 处理器

本手册主要涵盖最近出现的 IA-32 处理器，包括 Pentium®处理器、P6 系列处理器、Pentium 4 处理器和 Intel® Xeon™处理器。P6 系列处理器是指基于 P6 微架构的 IA-32 处理器，包括 Pentium Pro、Pentium Ⅱ、和 Pentium Ⅲ。Pentium 4 和 Intel Xeon 是基于 Intel® NetBurst™微架构的。

1.2. 《第 3 卷：系统开发指南》简介

本手册包括以下内容：

第 1 章 导读。介绍三卷《IA-32 Intel 架构软件开发人员手册》的内容和手册中使用的符号约定，罗列了 Intel 公司提供的相关手册和文档，供感兴趣的程序员和硬件设计人员进一步参考。

第 2 章 系统架构概况。描述了 IA-32 处理器的运行模式和对操作系统的支持机制，

包括面向系统的寄存器和数据结构以及面向系统的指令，讲述了实地址模式和保护模式互相切换所需的步骤。

第 3 章 保护模式的内存管理。描述了与分段和分页相关的数据结构、寄存器及指令，并介绍它们是如何用于实现“平坦”（未分段）的内存模型或者分段的内存模型。

第 4 章 保护。描述了 IA-32 架构对页保护和段保护所提供的支持，也介绍了特权规则、栈切换、指针合法性检查、用户态和管理态等的实施。

第 5 章 中断和异常处理。描述了 IA-32 架构定义的中断机制，介绍了中断和异常是如何与保护发生关系以及架构是如何处理各种异常的，并在本章末尾给出了各种异常的参考信息。

第 6 章 任务切换。描述了 IA-32 架构对多任务和任务之间保护的支持机制。

第 7 章 多处理器管理。描述了支持多处理器进行内存共享、内存访问排序和超线程技术的指令与标志。

第8章 高级可编程中断控制器（APIC）。描述了本地APIC的编程接口，并简要介绍了本地APIC与I/O APIC之间的接口。

第 9 章 处理器管理和初始化。描述了 IA-32 处理器在复位（Reset）初始化之后的状态，介绍了如何设置 IA-32 处理器以进入实地址模式和保护模式，和如何在两者之间进行切换。

第 10 章 内存高速缓存控制。描述了高速缓存的基本概念和 IA-32 架构支持的高速缓存机制，介绍了内存类型范围寄存器（MTRRs）及如何利用它们进行映射物理内存的内存类型，同时也介绍了如何使用 Pentium Ⅲ、Pentium 4 和 Intel Xeon 处理器引入的新的高速缓存控制和内存流化指令。

第 11 章 Intel® MMX™技术系统编程。描述了系统编程时需要考虑和处理 MMX 技术的几个方面：任务切换、异常处理、与现存系统环境兼容等。

第 12 章 SSE、SSE2 和 SSE3 系统编程。描述了系统编程时需要考虑和处理 SSE/SSE2/SSE3 扩展的几个方面：任务切换、异常处理、与现存系统环境兼容等。

第 13 章 系统管理。描述了 IA-32 架构的系统管理态（SMM）和热量（thermal）监测装置。

第 14 章 机器检测架构。描述了机器检测架构。

第 15 章 调试和性能监测。描述了 IA-32 架构中的调试寄存器和其它调试机制，并介绍了时间戳计数器和性能监测计数器。

第16章 8086仿真。描述了IA-32架构的实地址模式和虚拟8086模式。

第17章 混合使用16位和32位代码。描述了如何在同一程序或者任务中混合使用16位和32位代码模块。

第18章 IA-32架构的兼容性。描述了IA-32处理器之间的兼容性，包括Intel 286、Intel 386、Intel 486、Pentium、P6系列、Pentium 4和Intel Xeon处理器。32位IA-32处理器之间的差异，包括与此架构相关的一些专有特征，在这三卷《IA-32软件开发人员手册》中都有论述。本章汇总了与所有IA-32处理器兼容性相关的信息，并描述了与16位IA-32处理器（Intel 8086和Intel 286处理器）的基本差异。

附录A 性能监测事件。列出了可以用性能监测计数器计数的事件以及用于选择这些事件的代码，同时也描述了Pentium和P6系列处理器的性能检测事件。

附录B 模型相关寄存器(MSR)。列出了Pentium、P6系列、Pentium 4和Intel Xeon处理器中的MSR，并描述了它们的功能。

附录C P6系列处理器的MP初始化。举例说明在多MP系统中如何使用MP协议引导P6系列处理器。

附录D LINT0和LINT1输入编程。举例说明如何使用LINT0和LINT1引脚进行特定的中断向量编程。

附录E 机器检查错误代码的意义。举例说明如何解释P6系列处理器的机器检查错误代码。

附录F APIC总线消息格式。描述了P6系列和Pentium处理器的APIC总线上进行消息传递的消息格式。

1.3.《第1卷：基础架构》简介[1]

《基础架构》的内容如下：

第1章 导读。介绍了三卷《IA-32架构软件开发人员手册》的内容和手册中使用的符号约定，罗列了Intel公司提供的相关手册和文档，供感兴趣的程序员和硬件设计人员进一步参考。

第2章 IA-32架构概况。本章介绍了IA-32架构和基于此架构的Intel处理器系列，简要介绍这些处理器共有的特征以及IA-32架构的发展简史。

[1] 摘自《第1卷：基础架构》

第 3 章 基本运行环境。介绍了内存组织模型和应用程序使用的寄存器集。

第 4 章 数据类型。描述了处理器识别的数据类型和寻址方式，简要介绍了实数和浮点数格式以及浮点异常。

第 5 章 指令集汇总。列出了所有 IA-32 架构的指令，并根据指令技术类别进行了分组（通用、x87FPU、Intel MMX 技术、SSE、SSE2、SSE3 和系统指令），组内指令按照相关功能分类进行说明。

第 6 章 过程调用、中断和异常。描述了过程栈、过程调用机制以及服务于中断和异常的机制。

第 7 章 通用指令编程。描述了基于基本数据类型、通用寄存器和段寄存器的用于装载和保存、程序控制、数学和字符串等的指令，同时也描述了运行在保护模式下的系统指令。

第 8 章 x87 浮点单元编程。描述了包括浮点寄存器和数据类型在内的 x87 浮点单元（FPU）、浮点指令集和处理器浮点异常的产生条件。

第 9 章 Intel MMX 技术编程。描述了包括 MMX 寄存器和数据类型在内的 Intel MMX 技术和 MMX 指令集。

第 10 章 SIMD 扩展（SSE）编程。描述了包括 XMM 寄存器、MXCSR 寄存器和组合单精度浮点数据类型在内的 SSE 扩展、SSE 指令集和编写代码访问 SSE 扩展的方法。

第 11 章 SIMD 扩展 2（SSE2）编程。描述了包括 XMM 寄存器和组合双精度浮点数据类型在内的 SSE2 扩展、SSE2 指令集和用指令访问 SSE2 扩展的方法，本章也描述了 SSE 和 SSE2 指令产生的 SIMD 浮点异常，同时还给出了如何在操作系统和应用程序代码中统一支持 SSE 和 SSE2 扩展的一般原则。

第 12 章 SIMD 扩展 3（SSE3）编程。描述了 SSE3 扩展、SSE3 指令集和用指令访问 SSE3 的方法。

第 13 章 输入/输出。描述了处理器的 I/O 机制，包括 I/O 端口寻址、I/O 指令、I/O 保护机制。

第 14 章 处理器及其特征识别。描述如何识别 CPU 类型和处理器的特征。

附录 A EFLAGS 参考。汇总了 IA-32 指令是如何影响 EFLAGS 寄存器中的标志位的。

附录 B EFLAGS 条件码。汇总了条件代码指令中的条件跳转、传送和字节设置等是如何使用 EFLAGS 寄存器中的条件代码标志（OF、CF、ZF、SF、和 PF）的。

附录 C 浮点异常汇总。汇总了 x87FPU 浮点和 SSE/SSE2/SSE3 浮点指令产生的异常。

附录D 编写x87FPU异常处理程序指南。描述了如何设计和编写与MS-DOS兼容的FPU异常处理程序，包括软件和硬件需求以及汇编代码的例子，本附录也描述了编写健壮的FPU异常处理程序的基本技巧。

附录E 编写SIMD浮点异常处理程序指南。介绍了如何编写SSE/SSE2/SSE3浮点指令产生异常的处理程序。

1.4.《第2卷：指令集参考》简介[1]

《指令集参考》的内容如下：

第1章 导读。介绍了三卷《IA-32架构软件开发人员手册》的内容和手册中使用的符号约定，罗列了 Intel 公司提供的相关手册和文档，供感兴趣的程序员和硬件设计人员进一步参考。

第2章 指令格式。描述了所有IA-32指令在机器级的格式、许可的前缀编码、操作数标识符字节（ModR/M 字节）、寻址方式说明符字节（SIB 字节）、以及转移和立即数字节。

第3章 指令集参考，A-M。详细地逐条描述了IA-32指令，包括操作的规则描述、对各标志位的影响、对操作数和地址字节属性的影响以及可能产生的异常。这些指令是按照字母顺序进行排列的。一般指令、x87 FPU指令、MMX指令、SSE/SSE2/SSE3扩展指令和系统指令都包括在这里。

第4章 指令集参考，N-Z。

附录A 操作码映射。给出了IA-32指令的操作码映射。

附录B 指令格式和编码。给出了每条IA-32指令的二进制编码格式。

附录C Intel C/C++编译器等价功能。列出了Intel C/C++编译器的内部特征，以及与每个MMX、SSE/SSE2/SSE3指令等价的汇编指令序列。

1.5.符号约定

本手册对数据结构格式、指令的助记符、十六进制和二进制数字的表示都使用了特定的符号，了解这些约定就很容易阅读本手册。

[1] 摘自《第2卷：指令集参考》

1.5.1.位和字节顺序

在图示内存中的数据结构时，低地址部分位于图的底部，地址朝上增大，位的位置是从右至左进行排列的。一个给定位所代表的数值等于 2 的这个位的位置的幂。IA-32 处理器是“小尾巴”（little-endian）机器，这意味着一个字（共两个字节）的字节是从最低位开始的，图 1-1 说明这种约定。

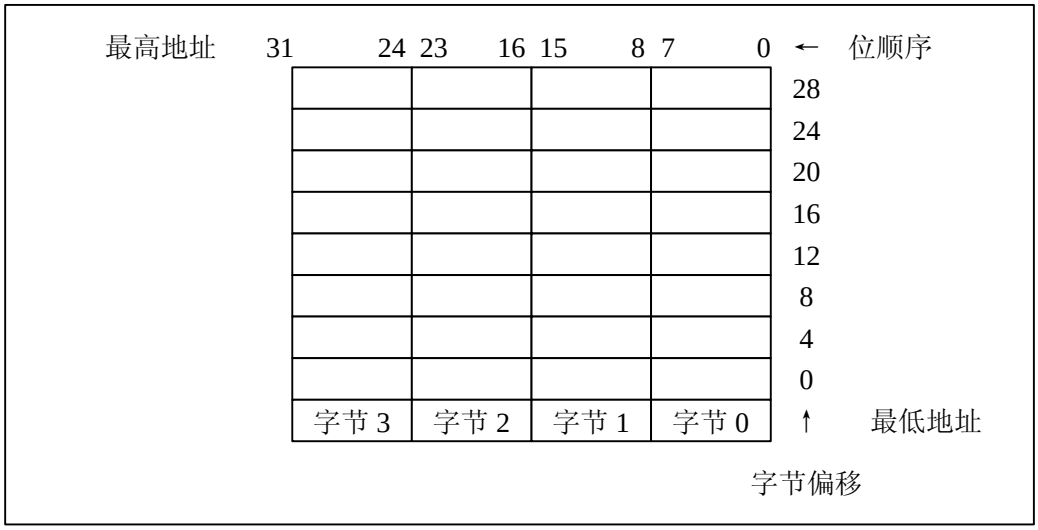

图 1-1 位与字节排列顺序

1.5.2.保留位与软件兼容性

在许多寄存器和内存布局描述中，某些位标记为“保留”（Reserved）。当有位标记为“保留”时，说明这些位是为将来处理器兼容而设的，软件在处理这些位时，要好比它将来要有某个值，虽然现在还不知道。修改保留位的后果并不能仅仅认为是未定义的，而是不可预测的。软件在涉及保留位时，应该遵守以下规定：

在测试包含保留位的寄存器的值时，不要依赖于任何保留位的状态，在测试之前应该把这些位屏蔽掉。

保存数据到内存或者寄存器时，不要依赖于任何保留位的状态。

不要依赖于保留位保存信息的能力。

当装载一个寄存器时，文档中如果对保留位的值有要求，就一定要装载这些值，或者就重新装载以前从同一寄存器读出的值。

注意

要避免软件依赖 IA-32 寄存器中的保留位的状态，依赖保留位就相当于让软件依赖处理器以一种不可预测的方式处理这些位，依赖保留位的程序有可能与将来的处理器不兼容。

1.5.3.指令操作数

当用符号来代表指令时，这里使用了 IA-32 汇编语言的一个子集。在这个子集中指令遵循以下格式：

标签（label）：助记参数 1、参数 2、参数 3

这里：

标签（label）是个标识符，后面紧跟着一个冒号。

助记符是与指令有着相同功能的保留字。

操作数参数 1、参数 2、参数 3 是可选的，根据操作码的不同可能有 0-3 个参数。有参数时，它或采用文字或采用标识符来代表数据项。操作数标识符或是寄存器保留字或是其它程序中声明的被赋值的数据项（本例中没有这部分说明）。

当算术或逻辑指令中有两个操作数时，右边的操作数是源操作数，左边的是目的操作数。例如：

LOADREG：MOV EAX，SUBTOTAL

在本例中，LOADREG 是一个标签，MOV 是指令助记符，EAX 是目的操作数，SUBTOTAL 是源操作数，在有些汇编语言中这个顺序正好相反。

1.5.4.十六进制和二进制数

基 16 数字（十六进制）是用十六进制数字表示的，后面跟有一个字母 H（比如 F82EH）。一个十六进制数字是下面字母中的一个：

0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、和 F。

基 2 数字（二进制）是用 1 和 0 的数字串来表示的，有时后面跟有一个字母 B（比如 1010B），这个“B”只在可能会引起混淆的情况下使用。

1.5.5. 分段寻址

处理器是按字节编址的，这意味着内存是按照着字节顺序进行组织和访问的。在访问一个或多个字节时，用字节地址进行定位它（们）在内存中的位置。内存可以被访问的范围就叫作地址空间。

处理器也支持分段寻址。这种寻址方式是指程序可以有多个独立的地址空间，叫作段，比如一个程序可以把它的指令和堆栈分别保存在独立的段中。代码地址总是指向代码段，堆栈地址总是指向栈空间。当访问段中的地址中，采用下面的方式：

段寄存器：字节地址

例如，下面的段地址标示 DS 寄存器指向的段中的 FF79H 地址：

DS：FF79H。

下面段地址标示代码段中的指令地址。CS 寄存器指向代码段，EIP 寄存器包含着指令地址：

CS：EIP。

1.5.6. 异常

异常通常是指由指令引起的错误事件，例如除 0 就会引起一个异常。然而有些异常，比如断点，在其它条件下出现。有些类型的异常可能会有错误代码，该代码包含关于这个错误的额外信息。下面是一个异常和错误代码的表示方法：

#PF（错误代码）

这个例子指的是一个页故障异常，这时的错误代码报告了某种类型的故障。在某些条件下，产生错误代码的异常可能不会提供准确的代码，在这种情况下，错误代码就是 0，就像下面的一般保护异常：

#GP（0）

1.6. 相关文献

与 IA-32 处理器相关的文献资料在下面的 Intel 网站上：

http://developer.Intel.com/design/processor/

这个站点列出的有些文档可以在线查看，有些可以在线订购。文献资料首先是根

据 Intel 处理器类别，然后是根据下面的类型列出的，即应用程序注意事项，数据表格、手册、论文和更新说明等栏目。

下面的文献可供参考：

- 某种 IA-32 处理器的数据表格
- 某种 IA-32 处理器的更新说明
- AP-485，《Intel 处理器识别和 CPUID 指令》，订购号：241618
- 《Intel® Pentium® 4 and Intel® Xeon™处理器优化参考手册》，订购号：248966

第 2 章 系统架构概况

IA-32 架构（从 Intel 386 处理器系列开始）为操作系统和系统开发软件提供了广泛的支持。这些支持是 IA-32 系统级架构的一部分，主要包括下列特性：

- 内存管理
- 软件模块保护
- 多任务
- 异常和中断处理
- 多处理器技术
- 高速缓存管理
- 硬件资源和电源管理
- 调试和性能监测

本章介绍 IA-32 系统架构和用来设置和控制处理器的系统寄存器，并简要说明处理器系统级（操作系统）指令。

只有系统开发人员才会使用 IA-32 系统级架构的特点，但是，为了给应用程序创建安全可靠的环境，应用程序开发人员也应该阅读本章和后续各章。

注意

本章和后续各章重点关注 IA-32 的“原生”或者说是保护模式操作。正如第 9 章“处理器管理和初始化”所述，所有 IA-32 处理器在加电或者复位后首先进入实地址模式，然后，必须由软件进行由实地址模式到保护模式下的切换。

2.1. 系统级架构概况

IA-32 系统级架构是由寄存器、数据结构和指令组成，这些指令是用来支持基本的系统级操作的，比如内存管理、中断和异常处理、任务管理和多处理器控制（多处理器技术）。图 2-1 大致列出了系统寄存器和数据结构。

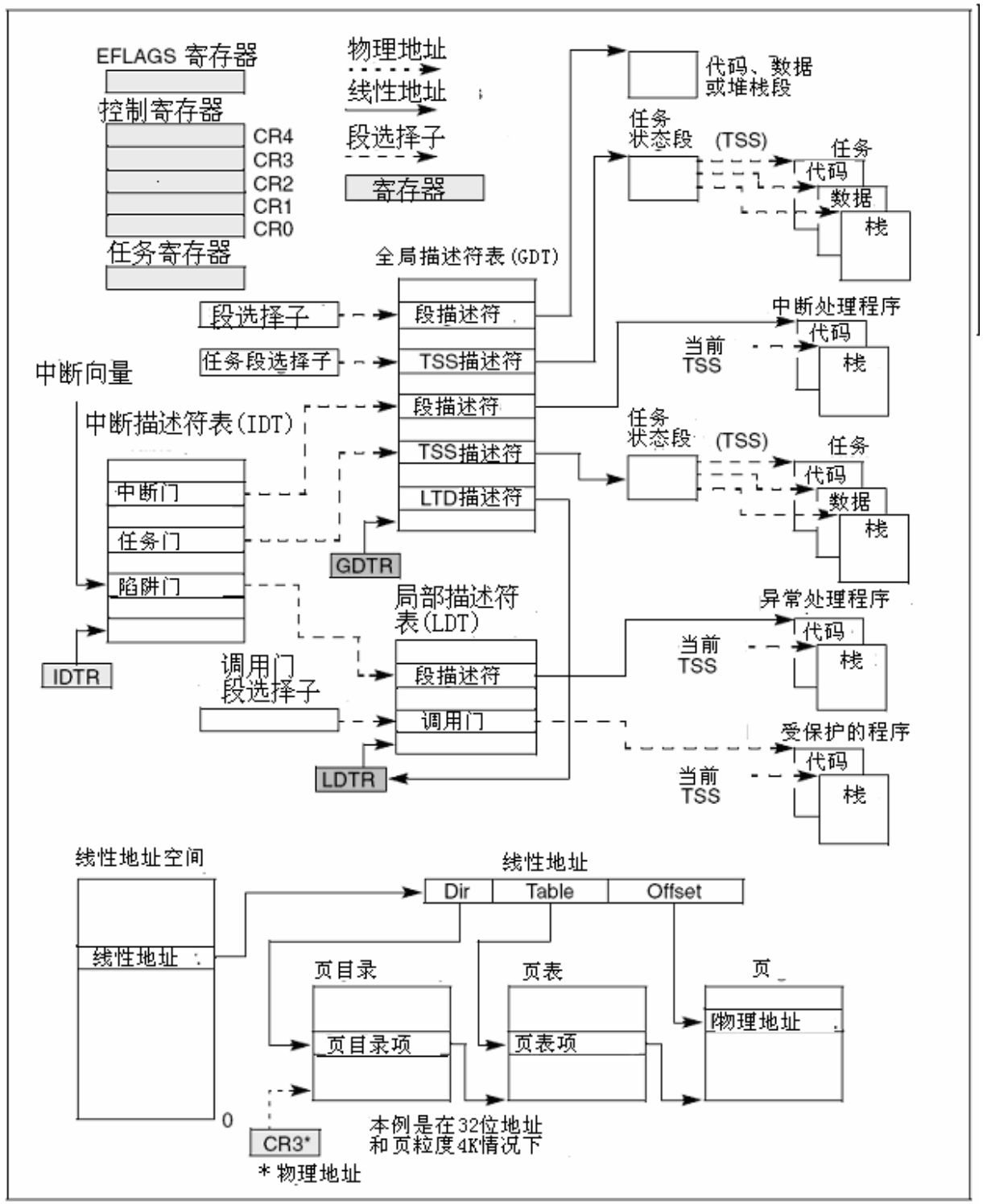

图2-1 IA-32系统级寄存器和数据结构

2.1.1.全局和局部描述符表

在保护模式操作时，所有内存访问要么通过全局描述符表（GDT - Global Descriptor Table）要么通过局部描述符表（LDT - Local Descriptor Table）（可

选），如图 2-1 所示。描述符表里的项叫做段描述符。一个段描述符包含一个段的基地址、访问权限、类型和用法信息。每个段描述符都有一个与之相关的段选择子。段选择子包含一个对 GDT 或 LDT（与它相关的段描述符）的索引、一个全局/局部标志（决定段选择子是指向 GDT 还是指向 LDT）和访问权限信息。

要想访问段中的一个字节，必须同时提供一个段选择子和一个偏移。段选择子提供对段的段描述符（在 GDT 或者 LDT 中）的访问。处理器从段描述符中获取段在线性地址空间里的基地址，偏移确定字节相对于基地址的地址。可以通过这种机制来访问在 GDT 或 LDT 中的各种合法代码、数据或者堆栈段，假定处理器运行的当前特权级（CPL - Current Privilege Level）可以访问这个段的话。CPL 被定义为当前执行代码段的保护级别。

图 2-1 中，实心箭头代表线性地址，虚线箭头代表段选择子，点划线箭头代表物理地址。为了简化起见，许多段选择子就直接指向段。然而实际上从段选择子到相应的段都是通过 GDT 或者 LDT。

GDT 的线性地址存放在 GDT 寄存器（GDTR）中，LDT 的线性地址存放在 LDT 寄存器（LDTR）中。

2.1.2. 系统段、段描述符和门

构成进程运行环境的除了代码、数据和堆栈段之外，系统架构还定义了两个系统段：任务状态段（TSS - Task-State Segment）和 LDT（GDT 不被看作段，因为它不是通过段选择子和段描述符访问的）。每种段都由一个段描述符来定义。

IA-32 系统架构也定义了一套称为门（调用门、中断门、陷阱门和任务门）的特殊描述符，以提供一种对不同于应用程序特权级的系统过程和处理程序的保护性访问途经。例如，通过对调用门的调用可以访问与当前代码段特权相同或者数字更低（特权更高）的代码段中的过程。通过调用门访问过程，调用程序必须提供调用门的选择子。然后，处理器比较 CPL 和调用门及调用门所指目的代码的特权级来检查访问权限。如果允许访问，处理器从调用门中取出目标代码段的选择子以及过程在目标代码段中的偏移。如果调用需要变换特权级，处理器就切换到那个级别的堆栈（新堆栈的段选择子是从当前任务的 TSS 中获得的）。调用门也有助于 16 位和 32 位代码段之间的互相切换。

2.1.3. 任务状态段和任务门

TSS（如图 2-1）定义任务执行环境的状态，包括通用寄存器、段寄存器、EFLAGS 寄存器、EIP 寄存器、段选择子、三个堆栈段（特权级 0、1、2 各一个堆栈）的指针等状态，以及与任务相关的 LDT 的选择子和页表的基地址。

保护模式下程序执行的所有状态，都放置在一个任务（称作当前任务）的场境（context）中。当前任务的 TSS 的段选择子保存在任务寄存器中。切换到一个任务的最简单的方法是调用或者跳转到那个任务。新任务的 TSS 的段选择子是通过 CALL 或 JMP 指令给出的。切换任务时，处理器要执行下列操作：

1、保存当前任务的状态到当前 TSS 中。

2、装载新任务的段选择子到任务寄存器中。

3、通过 GDT 中的段描述符访问新的 TSS。

4、将新 TSS 中新任务的状态装载到通用寄存器、段寄存器、LDTR、控制寄存器 CR3（页表基地址）、EFLAGS 寄存器和 EIP 寄存器中。

5、开始执行新任务。

也可通过任务门访问任务。任务门与调用门很相似，所不同的是它提供对 TSS（通过选择子）而不是对代码段的访问。

2.1.4. 中断和异常处理

外部中断、软件中断和异常是通过中断描述符表（IDT）处理的，如图 2-1 所示。IDT 中包含访问中断和异常处理程序的门描述符的集合。像 GDT 一样，IDT 不是一个段，IDT 的线性基地址包含在 IDT 寄存器中（IDTR）。

IDT 中的描述符可以是中断门、陷阱门或任务门。处理器必须先从内部硬件、外部中断控制器或者通过诸如 INT、INTO、INT 3、BOUND 指令收到一个中断向量（中断号），才去访问中断或异常处理程序。中断向量是 IDT 中门描述符的索引。如果选中的门描述符是中断门或者陷阱门，就如同通过调用门调用过程一样去访问相应的处理程序；如果是任务门，就通过任务切换访问其处理程序。

2.1.5. 内存管理

IA-32 系统架构既支持直接物理内存寻址也支持虚拟内存（通过分页）。采用物理寻址（模式）时，线性地址就被当作是物理地址；使用分页（模式）时，所有的代码、数据、堆栈和系统段、GDT、IDT 都可能被分页，且只有最近被访问过的页保留在物理内存中。

页（在 IA-32 架构中有时被称作页框）在物理内存中的位置保存在两类系统数据结构中（一张页目录表和一组页表），它们都驻留在物理内存中（参看图 2-1）。页目录表中的一个表项包含有一张页表的物理基地址、访问权限、内存管理信息，页表中的一个表项包含有页框的物理地址、访问权限和内存管理信息。页目录表的基地址保存在控制寄存器 CR3 中。

使用分页机制时，线性地址被分为三个部分，分别提供在页目录表、页表和页框中的偏移。

一个系统可以有一个或者多个页目录表，比如每个任务都可以有自己的页目录表。

2.1.6. 系统寄存器

为了更好地进行处理器初始化及控制系统的运行，IA-32 架构提供了几个系统寄存器和若干个在 EFLAGES 寄存器内的系统标志：

- EFLAGES 寄存器内的系统标志和 IOPL 域控制着任务和模式的切换、中断处理、指令跟踪和访问权限。2.3.“EFLAGES 寄存器中的系统标志和域”将对此进行描述。
- 控制寄存器（CR0、CR2、CR3、CR4）中有一些标志和数据域用于控制系统级操作，另外一些标志则专用来支持操作系统和管理程序。2.5.“控制寄存器”将对此进行描述。
- 调试寄存器（图 2-1 内没有列出）允许在调试软件和系统软件内设置断点。第 15 章“调试和性能监测”将对此进行描述。
- GDTR、LDTR 和 IDTR 寄存器内包含各自对应的表的线性地址和大小（界限）。2.4.“内存管理寄存器”将对此进行描述。
- 任务寄存器包含当前任务的 TSS 的线性地址和大小。2.4.“内存管理寄存器”

将对此进行描述。

- 模型相关的寄存器（图 2-1 内没有列出）。

模型相关寄存器（MSR）是一组主要用于操作系统和管理程序（代码运行在特权级 0）的寄存器，控制着如调试扩展、性能监测计数器、机器检测架构和内存类型范围（MTRR）等事项。

这些寄存器的个数和功能在 IA-32 架构系列的不同处理器中各有不同。9.4.“模型相关寄存器（MSR）”有更多论述，附录 B“模型相关寄存器（MSR）”中有 MSR 的完整列表。

大多数系统都限制应用程序访问系统寄存器（EFLAGES 寄存器除外），然而系统也可以设计成所有程序均运行在最高特权级（0 级）上，在这种情况下应用程序可以修改系统寄存器。

2.1.7.其它系统资源

除了前几节介绍的系统寄存器和数据结构之外，IA-32 系统架构还提供下列资源：

- 操作系统指令（参看 2.6.“系统指令汇总”）。
- 性能监测计数器（图 2-1 没有列出）。
- 内部高速缓存和缓冲区（图 2-1 没有列出）。

性能监测计数器是事件计数器，可对其编程来记录诸如解码指令的个数、接收的中断个数或者高速缓存的安装数量等。更详细的介绍参见 15.8.“性能监测概况”。

处理器提供了几个内部高速缓存和缓冲区。这些高速缓存用来保存数据和指令，缓冲区用来保存诸如系统和应用程序段的解码地址以及等待执行的写操作等。详细的介绍参见第 10 章“内存高速缓存控制”。

2.2.运行模式

IA-32 架构支持如下三种运行模式和一种准运行模式：

- **保护模式。**这是处理器的原生模式。在该模式下，处理器所有的指令和结构特点都是可用的，且性能最佳，能力最强。对所有新应用程序和操作系统推荐使用该模式。
- **实地址模式**。该模式提供 Intel 8086 处理器编程环境，并稍作扩展（比如切

换到保护模式或系统管理态的能力）。

- **系统管理态（SMM）**。系统管理态（SMM - System Management Mode）是所有 IA-32 处理器的一个标准架构特征，从 Intel 386SL 处理器开始引入的。它为操作系统和管理程序提供了一种透明的机制来实现电源管理和 OEM 的专有特征。SMM 通过激活外部系统中断引脚（SMI＃）产生一个系统中断（SMI）而进入。进入 SMM 时，处理器先保存好当前进程或任务的场境，然后切换到一个单独的地址空间。SMM 相关代码的执行是透明的。从 SMM 返回时，处理器就被置换到 SMI 之前的状态。
- **虚拟 8086 模式**。在保护模式中，处理器支持一种准运行模式叫作虚拟 8086 模式。该模式允许处理器在多任务的保护模式下执行 8086 程序。

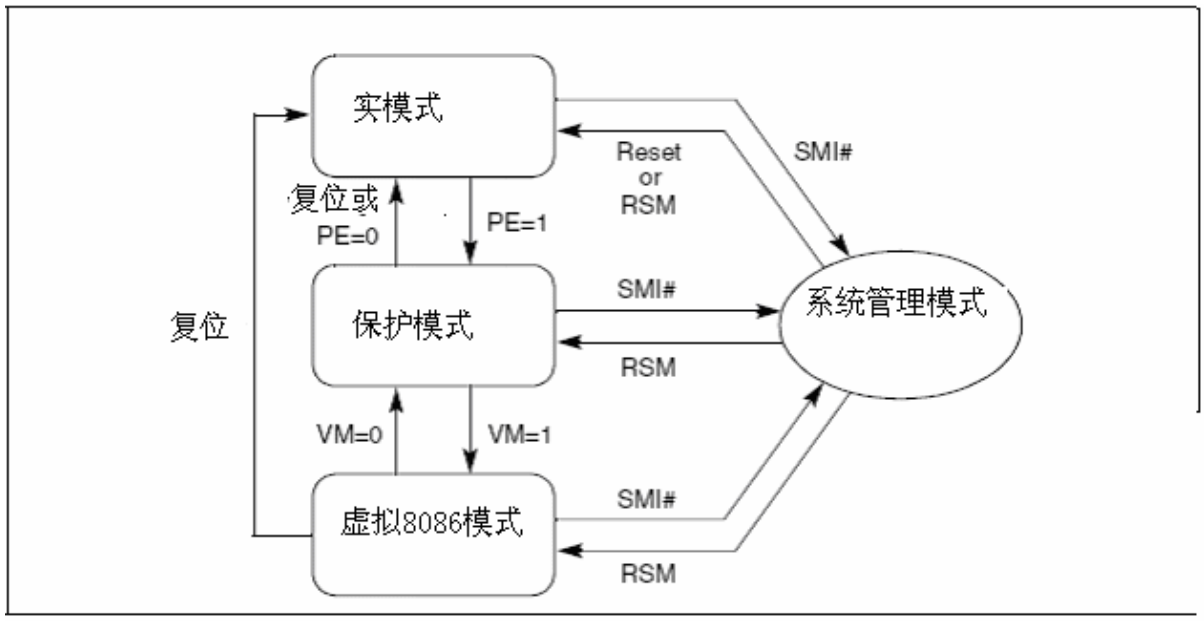

图2-2 处理器运行模式之间的转换

处理器在上电或复位后就进入到实地址模式，然后，由控制寄存器 CR0 中的 PE 标志决定处理器是运行在实地址模式下还是在保护模式下（参见 2.5.“控制寄存器”）。模式切换更详细的介绍参见 9.9.“模式切换”。

EFLAGES 寄存器中的 VM 标志决定着处理器运行在保护模式下还是虚拟 8086 模式下。保护模式和虚拟 8086 模式之间的切换是作为任务切换或从中断和异常处理程序返回的一部分（参见 16.2.5.“进入虚拟 8086 模式”）来处理的。

不论处理器是在实地址模式、保护模式还是虚拟 8086 模式下，只要接收到 SMI 就切换到 SMM 模式。一旦执行 RSM 指令，处理器就返回到 SMI 发生时的模式。

2.3.EFLAGES 寄存器中的系统标志和域

EFLAGES 中的系统标志和 IOPL 域用于控制 I/O、可屏蔽硬件中断、调试、任务切换和虚拟 8086 模式（见图 2-3）。只有特权代码（通常是操作系统或管理程序代码）可以修改这些位。

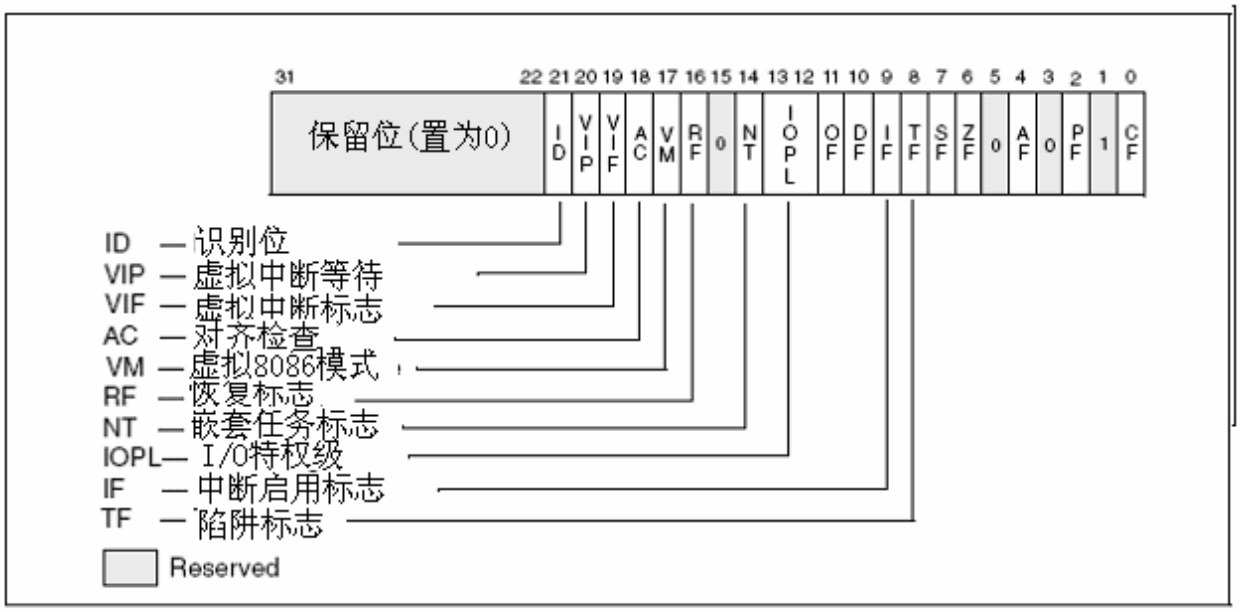

图2-3 EFLAGES寄存器中的系统标志

系统标志和 IOPL 的作用如下：

TF　**陷阱（第 8 位）。**置 1 则开启单步执行调试模式，置 0 则关闭单步执行。在单步执行模式下，处理器在每条指令后产生一个调试异常，这样在每条指令执行后都可以查看执行程序的状态。如果程序用 POPF、POPFD 或者 IRET 指令设置 TF 标志，那么这之后的第一条指令就会产生调试异常。

IF　**中断许可（第 9 位）。**控制处理器对可屏蔽硬件中断（见 5.3.2.“可屏蔽硬件中断”）请求的响应。置 1 则开启可屏蔽硬件中断响应，置 0 则关闭可屏蔽硬件中断响应。IF 标志不影响异常和不可屏蔽中断（NMI）的产生。CPL、IOPL 和控制寄存器 CR4 中的 VME 标志决定着 IF 标志是否可由 CLI、STI、POPF、POPFD 和 IRET 指令修改。

IOPL　**I/O 特权域（第 12 位和第 13 位）。**标示当前进程或任务的 I/O 特权级别。当前进程或任务的 CPL 必须小于或等于 IOPL 才可以访问 I/O 地址空间。只有 CPL 为 0 的进程才可用 POPF 和 IRET 指令修改这个域。《基本架构》

的第 12 章“输入/输出”对 IOPL 和 I/O 操作之间的关系有详细的介绍。

IOPL 是控制 IF 标志修改的机制之一，同时也是当虚拟模式扩展生效时（控制寄存器 CR4 中的 VME 置位），控制虚拟 8086 模式下中断处理的机制之一。

NT **嵌套任务（第 14 位）。**控制被中断和被调用的任务的嵌套执行链。处理器调用一个由 CALL 指令、中断或者异常触发的任务时设置该位，调用 IRET 指令返回时检测并修改该位。该标志可以由 POPF/POPFD 指令直接置位或置 0，然而在应用程序中修改该标志的状态会产生不可预料的异常。

参见 6.4.“任务链”对嵌套任务的详细描述。

RF **恢复（第 16 位）。**控制处理器对指令断点的响应。置 1 则暂时禁用指令断点产生调试异常（＃DE），但是其它异常情况仍可以产生异常。置 0 则指令断点产生调试异常。

RF 标志的主要功能是许可从调试异常（指令断点引发的）后面的那个指令开始继续执行。调试软件必须在用 IRETD 指令返回到被中断程序之前，将栈中的 EFLAGES 映象中的该位置为 1，以阻止指令断点产生另外的调试异常。在返回并成功执行断点指令之后，处理器会自动清零该位，从而许可继续产生指令断点故障。

详细介绍请参见 15.3.1.1.“指令断点异常”。

VM **虚拟 8086 模式（第 17 位）。**置 1 则进入虚拟 8086 模式，置 0 则返回保护模式。参见 16.2.1.“启用虚拟 8086 模式”里的详细介绍。

AC **对齐检查（第 18 位）。**置位该标志和控制寄存器 CR0 的 AM 标志则启用对内存引用的对齐检查，清除这两个标志则禁用对齐检查。当引用一个没有对齐的操作数时，将会产生一个对齐检查异常，比如在奇地址引用一个字地址或在不是 4 的倍数的地址引用一个双字地址。对齐检查异常只在用户态（3 级特权）下产生。默认特权为 0 的内存引用，比如段描述符表的装载，并不产生这个异常，尽管同样的操作在用户态会产生异常。

对齐检查异常用于检查数据的对齐，当处理器之间交换数据时这很有用，交换数据需要所有的数据对齐。对齐检查异常也可供解释程序使用。让某些指针不对齐就好比做上特殊标记，这样就无需对每个指针都进行检查，只在用到的时候，对这些特殊指针进行处理就可以了（一旦用到这些特殊

指针，因为不对齐，系统就会产生对齐异常，这样就可以在对齐异常处理程序中放置相应的处理方法——译者）。

VIF **虚拟中断（第 19 位）。**是 IF 标志的一个虚拟映象。这个标志是和 VIP 标志一起使用的。当控制寄存器 CR4 中的 VME 或者 PVI 标志置为 1 且 IOPL 小于 3 时，处理器只识别 VIF 标志（VME 标志用来启用虚拟 8086 模式扩展，PVI 标志启用保护模式下的虚拟中断）。

详细内容参见 16.3.3.5.“方法 6：软件中断处理”和 16.4.“保护模式虚拟中断”。

VIP **虚拟中断等待(pending)（第 20 位）。**置 1 表明有一个正在等待处理的中断，置 0 表明没有等待处理的中断。该标志和 VIF 一起使用。处理器读取该标志但从来不修改它。当 VME 标志或者控制寄存器 CR4 中的 PVI 标志置 1 且 IOPL 小于 3 时，处理器只识别 VIP 标志。（VME 标志启用虚拟 8086 模式扩展，PVI 标志启用保护模式虚拟中断）。

详细内容参见 16.3.3.5“方法 6：软件中断处理”和 16.4.“保护模式虚拟中断”。

ID **识别（第 21 位）。**置 1 或 0 表明是否支持 CPUID 指令。

2.4. 内存管理寄存器

处理器用 4 个内存管理寄存器（GDTR、LDTR、IDTR 和 TR）来确定控制分段内存管理的数据结构的所在位置（如图 2-4）。有专门的指令来装载和保存这些寄存器。

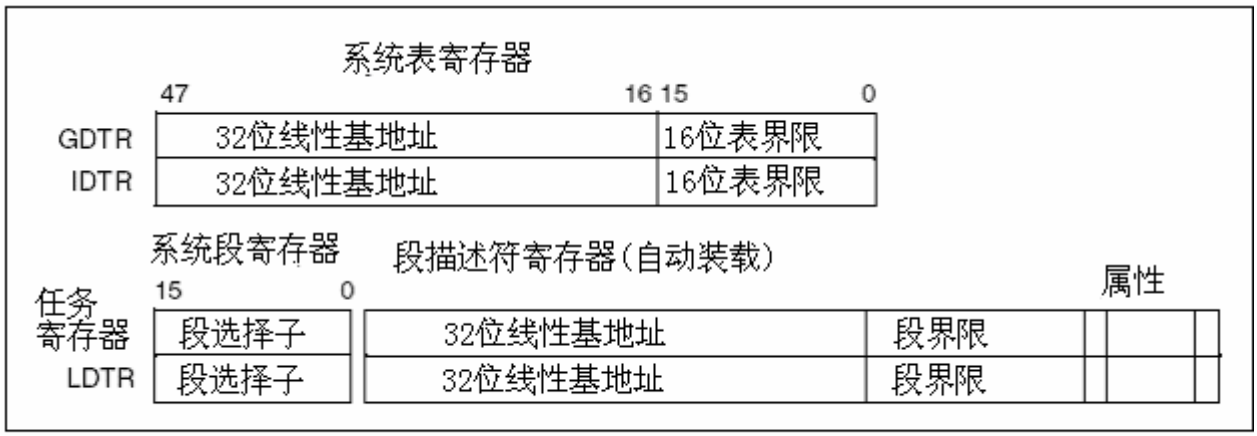

图2-4 内存管理寄存器

2.4.1. 全局描述符表寄存器（GDTR）

GDTR 寄存器保存 GDT 的 32 位基地址和 16 位表界限。基地址是指 GDT 的第一个字节（字节 0）的线性地址，表界限是指表中的字节个数。LGDT 和 SGDT 指令分别用来装载和保存 GDTR 寄存器。处理器一上电或复位，基地址就被设为缺省值 0，表界限设为 FFFFH。作为处理器进入保护模式初始化过程的一部分，一个新的基地址必须装入 GDTR。详细说明参看 3.5.1.“段描述符表”。

2.4.2. 局部描述符表寄存器（LDTR）

LDTR 寄存器保存 16 位段选择子、32 位基地址、16 位段界限和 LDT 属性。基地址是指 LDT 段的第一个字节（字节 0）的线性地址，段界限是指段中的字节个数。参见 3.5.1.“段描述符表”对于基地址和界限域的详细说明。

LLDT 和 SLDT 指令是专门用来装载和保存 LDTR 寄存器段选择子那部分的。LDT 所在段必须在 GDT 中有一个段描述符。当 LLDT 指令装载一个段选择子到 LDTR 中时，LDT 描述符的基地址、界限和描述符属性就自动装载到 LDTR 中。

当进行任务切换时，新任务的段选择子和描述符就自动被装载进 LDTR 中。在新的信息装载到 LDTR 寄存器前，LDTR 的内容不会被自动的保存。

处理器上电或复位时，LDTR 寄存器的段选择子和基地址都被设缺省值 0，界限被设为 FFFFH。

2.4.3. 中断描述符表寄存器（IDTR）

IDTR 寄存器保存 IDT 的 32 位基地址和 16 位表界限。基地址是指 IDT 的第一个字节（字节 0）的线性地址，表界限是指表中的字节个数。LIDT 和 SIDT 是专门用来装载和保存 IDTR 寄存器的指令。处理器上电或复位时，基地址被设为缺省值 0，界限被设为 FFFFH。作为处理器初始化过程的一部分，寄存器中的基地址和界限可以改变。参见 5.10.“中断描述符表（IDT）”关于基地址和界限域的详细说明。

2.4.4.任务寄存器（TR）

任务寄存器保存 16 位的段选择子、32 位基地址、16 位段界限和当前任务的 TSS 属性。它引用 GDT 中的 TSS 描述符。基地址指明 TSS 的第一个字节（字节 0）的线性地址，段界限确定 TSS 的字节个数。（参见 6.2.3.“任务寄存器”有关详细描述。）

LTR 和 STR 指令分别用来装载和保存任务寄存器段选择子那部分的。当用 LTR 装载一个段选择子到任务寄存器时，从相关 TSS 描述符中取出的基地址、界限和描述符属性都会被自动的装载到任务寄存器。处理器上电或复位时，基地址设成默认值 0，界限被设成 FFFFH。

进行任务切换时，任务寄存器就自动装载新任务的段选择子和 TSS 描述符。在新任务的信息写入任务寄存器之前，任务寄存器中原来的内容不会自动被保存。

2.5.控制寄存器

控制寄存器（CR0、CR1、CR2、CR3 和 CR4，见图 2-5）决定着处理器的运行模式和当前正在执行任务的特征，具体如下：

- CR0——包含系统控制标志，这些标志控制着处理器的运行模式和状态。
- CR1——保留。
- CR2——包含页故障的线性地址（引起页故障的线性地址）。
- CR3——包含页目录表的物理基地址和二个标志（PCD 和 PWT）。该寄存器也被称为页目录基地址寄存器（PDBR）。页目录表的基地址由高 20 位确定，低 12 位是 0，所以页目录表的地址必须是页边界对齐的（4K 字节）。PCD 和 PWT 标志控制着页目录表在处理器内部数据高速缓存中的高速缓存(但不控制页目录信息的 TLB 高速缓存)。当使用物理地址扩展时，CR3 寄存器包含页目录指针表的基地址（见 3.8.“使用 PAE 分页机制的 36 位物理寻址”）。
- CR4——包含一组标志，这些标志启用 IA-32 系统架构上的几个扩展，指示操作系统或管理程序支持处理器的特殊功能。控制寄存器可以用 MOV 指令“从寄存器读或者写到寄存器”的方式来进行读取或者装载（修改）。在保护模式下，MOV 指令允许读取或者装载控制寄存器（只在 0 级特权下）。这个限制意味着应用程序或者操作系统过程（运行在 1、2、3 级特权下）不能读取或者装载控

制寄存器。

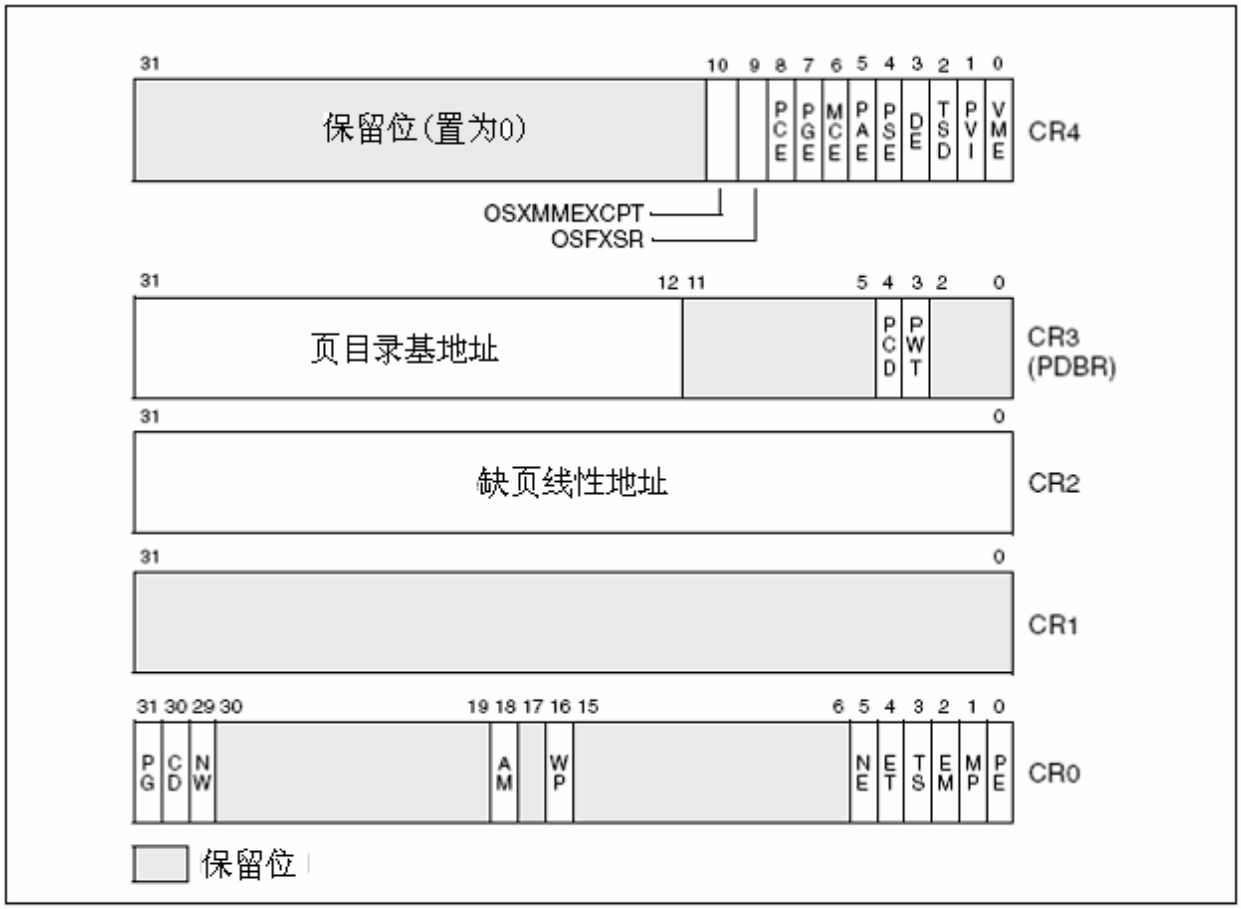

图2-5. 控制寄存器

装载控制寄存器时，保留位应该保持以前读取的值。控制寄存器中标志的作用如下：

PG **分页（CR0 的第 31 位）。**置 1 启用分页，置 0 不启用分页。当禁用分页时，所有的线性地址都当作物理地址对待。如果 PE 标志（CR0 的第 0 位）没有置 1，PG 标志将不起作用。实际上，如果在 PE 标志为 0 的情况下，将 PG 标志置 1 会产生一个一般保护异常（#GP）。见 3.6.“分页（虚拟内存）概况”对处理器分页机制的详细说明。

CD **禁用高速缓存（CR0 的第 30 位）。**当 CD 和 NW 标志置为 0 时，则开启处理器内部（和外部）高速缓存对整个物理内存位置的进行高速缓存。当 CD 标志置 1 时，则禁用高速缓存，如表 10-5 所示。为阻止处理器访问和修改它的高速缓存，必须将 CD 标志置 1 并且使高速缓存失效，

这样高速缓存就不会被命中（见 10.5.3.“阻止高速缓存”）。10.5.“高速缓存控制”有对选中的页或者内存区域的高速缓存的其它限制的详细描述。

NW **不直写（CR0 的第 29 位）。**当 NW 和 CD 标志都置 0 时，则开启命中高速缓存的写操作的回写（即 write-back，对 Pentium 4、Intel Xeon、P6 系列和 Pentium 处理器而言）或直写（即 write-through，对 Intel486 处理器而言），启用失效周期。表 10-5 详细介绍了 CD 和 NW 各种设置时 NW 标志对高速缓存的影响。

AM **对齐屏蔽（CR0 的第 18 位）。**置 1 则启用自动对齐检查，置 0 则禁用对齐检查。对齐检查只有当 AM 标志和 EFLAGES 中的 AC 标志都置 1、CPL 是 3、处理器运行在保护模式或者虚拟 8086 模式下才进行。

WP **写保护（CR0 的第 16 位）。**置 1 则禁止管理级程序写用户级的只读页，置 0 则允许管理级程序写用户级的只读页。这个标志是用来在创建（forking）一个新进程时协助实现写时复制（COW——copy on write）的，在 UNIX 操作系统中就是如此。

NE **数值错误（CR0 中的第 5 位）。**置 1 则启用原生的（内部的）x87 FPU 错误报告机制，置 0 则启用类 PC 的 x87 FPU 错误报告机制。当 NE 标志置 0 且 IGNNE#输入被激活时，忽略 x87 FPU 错误；当 NE 标志置 0 且 IGNNE#输入未被激活时，一个未屏蔽的 x87 FPU 错误将导致处理器激活 FERR#引脚产生一个外部中断，并且立即停止指令的执行，直到执行下一条等待中的浮点指令或 WAIT/FWAIT 指令为止。FERR#引脚是用来产生一个输入信号送到外部中断控制器的（FERR＃引脚是模拟 Intel 287 和 Intel 387DX 数学处理器的 ERROR#引脚的）。NE 标志、IGNNE＃引脚和 FERR＃引脚是用于外部逻辑（器件）以实现类 PC 的错误报告机制。（参见第 8 章“软件异常处理”和《基本架构》附录 D 中关于 x87 FPU 错误报告机制，以及更多关于 FERR＃引脚被激活时的详细说明，这与它的具体实现密切相关。）

ET **扩展类型（CR0 的第 4 位）。**在 Pentium 4、Intel Xeon、P6 系列、和 Pentium 处理器中这是保留位，被硬编码为 1。在 Intel 386 和 Intel

486 处理器中，这个标志置 1 表示对 Intel 387DX 数学协处理器指令的支持。

TS **任务切换（CR0 的第 3 位）。**任务切换时，允许延迟至新任务实际执行 x87 FPU、MMX、SSE、SSE2、SSE3 指令时，再保存 x87 FPU、MMX、SSE、SSE2、SSE3 的场境。每当任务切换时处理器设置该位，当执行 x87 FPU、MMX、SSE、SSE2、SSE3 指令时测试该位。

如果 TS 标志置 1 并且 EM 标志（CR0 的第 2 位）置 0，那么执行任何 x87 FPU、MMX、SSE、SSE2、SSE3 指令之前，都会产生“设备不可使用”异常（＃NM），但是这些指令不包括 PAUSE、PREFETCHh、SFENCE、LFENCE、MFENCE、MOVNTI 和 CLFLUSH。（参见下面 WAIT/FWAIT 指令特殊情况的叙述。）

- 如果 TS 标志置 1 并且 MP 标志（CR0 的第 1 位）和 EM 标志都置 0，那么在执行 x87 FPU WAIT/FWAIT 指令之前不会产生＃NM 异常。
- 如果 EM 标志置 1，TS 标志的值对 x87 FPU、MMX、SSE、SSE2、SSE3 指令的执行没有什么影响。

表 2-1 列出了处理器遇到 x87 FPU 指令时，根据 TS、EM、MP 标志的不同设置所作出的不同反应。表 11-1 和 12-1 列出了处理器分别遇到 MMX 和/或 SSE、SSE2、SSE3 指令时所作出的反应。

处理器在进行任务切换时并不会自动保存 x87 FPU、XMM 和 MXCSR 寄存器的内容。相反它将 TS 标志置为 1，这样在新任务的指令流中，无论处理器何时遇到 x87 FPU、MMX、SSE、SSE2 或 SSE3 指令（前面列出的指令除外），都会引起＃NM 异常。

＃NM 故障处理程序清除 TS 标志（用 CLTS 指令），保存 x87 FPU、XMM 和 MXCSR 寄存器的场境。如果任务从未遇到 x87 FPU、MMX、SSE、SSE2、SSE3 指令，那么 x87 FPU、MMX、SSE、SSE2、SSE3 的场境就从不保存。

表 2-1 根据 EM、MP 和 TS 不同组合，x87 FPU 指令动作

CR0标志			X87指令类型	
EM	MP	TS	浮点	WAIT/FWAIT
0	0	0	执行	执行
0	0	1	＃NM异常	执行

0	1	0	执行	执行
0	1	1	＃NM异常	＃NM异常
1	0	0	＃NM异常	执行
1	0	1	＃NM异常	执行
1	1	0	＃NM异常	执行
1	1	1	＃NM异常	＃NM异常

EM **仿真（CR0 的第 2 位）。**置 1 则表明处理器没有内部或者外部的 x87 FPU，置 0 则表明有 x87 FPU。这个标志也影响 MMX、SSE、SSE2、SSE3 指令的执行。当 EM 为 1 时 x87 指令的执行会产生一个“设备不可使用”异常(#NM)。当处理器没有 x87 FPU 或者没有连接到外部数学协处理器时，必须将该位置为 1。设置该位将强制所有的浮点指令由软件仿真。表 9-2 列出了根据 IA-32 处理器和 x87 FPU 或者系统中有的数学协处理器的情况推荐设置该标志的值。表 2-1 列出了 EM、MP 和 TS 标志的相互影响。

另外，当 EM 标志为 1 时，执行 MMX 指令将会产生非法操作码异常(#UD)(见表 11-1)。所以，如果 IA-32 处理器要想利用 MMX 技术，EM 标志必须设置为 0 以开启 MMX 指令的执行。

对于 SSE/SSE2/SSE3 扩展也是一样，当 EM 为 1 时，大多数 SSE/SSE2/SSE3 指令的执行都会产生一个非法操作码异常(#UD)(见表 12-1)。所以，如果 IA-32 处理器要想利用 SSE/SSE2/SSE3 扩展，EM 就必须置为 0 以开启这些扩展指令的运行。不受 EM 的值影响的 SSE/SSE2/SSE3 指令有：PAUSE、PREFETCHh、SFENCE、LFENCE、MFENCE、MOVNTI 和 CLFLUSH。

MP **监测协处理器(CR0 的第 1 位)。**控制 WAIT(或者 FWAIT)指令与 TS 标志(CR0 的第 3 位)的相互作用。如果 MP 标志是 1，当 TS 标志是 1 时，WAIT 指令会产生“设备不可使用”异常(#NM)。如果 MP 标志是 0，WAIT 指令忽略 TS 标志的值。根据 IA-32 处理器和系统中是否有 x87 FPU 或协处理器，表 9-2 列出了这个标志的推荐设置。表 2-1 列出了 MP、EM 和 TS 标志的相互作用。

PE **启用保护模式（CR0 的第 0 位)。**置 1 则启用保护模式，置 0 则启用

实地址模式。这个标志并不直接启用分页机制，只是启用了段级保护。要是启用分页机制，必须将 PE 和 PG 标志都设为 1。参见 9.9.“模式切换”利用 PE 标志在实模式与保护模式之间进行切换的详细介绍。

PCD **禁用页级高速缓存（CR3 的第 4 位）。**控制当前页目录表的高速缓存。置 1 则禁止页目录表的高速缓存，置 0 则启用页目录表的高速缓存。这个标志只影响处理器内部高速缓存（L1 和 L2 都存在的情况下）。如果没有启用分页机制(CR0 中的 PG 标志置 0)或者 CR0 中的 CD 标志置 0（禁用缓存），处理器忽略这个标志。第 10 章“内存高速缓存控制”中有使用这个标志的详细说明。参见 3.7.6.“页目录表项和页表项”对页目录表项和页表项伴侣 PCD 标志的详细说明。

PWT **页级透明写（CR3 的第 3 位）。**控制着当前页目录表的直写或回写的高速缓存策略。如果 PWT 标志置 1，则启用直写高速缓存，置 0 则启用回写高速缓存。这个标志只影响内部缓存（在 L1 和 L2 都存在的情况下），如果没有启用分页机制（CR0 中的 PG 标志为 0）或者 CR0 中的 CD 标志为 1（禁用缓存），处理器忽略这个标志。参见 10.5.“缓存控制”中对这个标志的详细介绍。参见 3.7.6.“页目录表项和页表项”对页目录表项和页表项伴侣 PCD 标志的详细说明。

VME **虚拟 8086 模式扩展（CR4 的第 0 位）。**置 1 则在虚拟 8086 模式下启用中断和异常处理扩展，置 0 则关闭。使用虚拟模式扩展能改进虚拟 8086 应用程序的性能，它取消了调用虚拟 8086 监控程序去处理 8086 程序执行中出现的中断和异常的累赘，而是把中断和异常重定向回 8086 程序的处理程序。它也是虚拟中断标志（VIF）的硬件支持，用来改进在多任务及多处理器环境中执行 8086 程序的可靠性。参见 16.3.“虚拟 8086 模式的中断和异常处理”关于使用这一特征的详细介绍。

PVI **保护模式虚拟中断（CR4 的第 1 位）。**置 1 则开启保护模式下虚拟中断标志（VIF）的硬件支持，置 0 则禁用保护模式下的 VIF 标志。参见 16.4.“保护模式虚拟中断”对这一特征用法的详细介绍。

TSD **禁用时间戳（CR4 的第 2 位）。**置 1 则限制 0 级特权程序执行 RDTSC 指令，置 0 则允许任何特权程序执行 RDTSC 指令。

DE **调试扩展（CR4 的第 3 位）。**置 1 则引用调试寄存器 DR4 和 DR5 引发一个未定义操作码(#UD)异常，置 0 时，为保持与在早期 IA-32 处理器上写的程序兼容，处理器需以别名引用 DR4 和 DR5。更详细说明的参见 15.2.2.“调试寄存器 DR4 和 DR5”。

PSE **页尺寸扩展(CR4 的第 4 位)。**置 1 则页大小为 4M 字节，置 0 则页大小为 4K 字节。参见 3.6.1.“分页选项”对这个标志用法的介绍。

PAE **物理地址扩展(CR4 的第 5 位)。**置 1 则启用引用 36 位物理地址的分页机制，置 0 时只可引用 32 位地址。参见 3.8.“使用 PAE 分页机制实现 36 位物理寻址”中对物理地址扩展的详细说明。

MCE **启用机器检测(CR4 的第 6 位)。**置 1 则启用机器检测异常，置 0 则禁用。参见第 14 章“机器检测架构”对机器检测异常和机器检测架构的详细说明。

PGE **启用全局页(CR4 的第 7 位)。**(P6 系列处理器中引入的。)置 1 则启用全局页，置 0 则禁用。全局页特性允许把那些经常被使用或共享的页标记为全局的(通过页目录或者页表项中的第 8 位——全局标志来实现)，以供所有用户使用。在任务切换或者写 CR3 寄存器时，不从 TLB 中刷新全局页。

启用全局页特性时，在设置 PGE 标志之前必须先启用分页机制(通过设置 CR0 中的 PG 标志)。颠倒这个顺序会影响程序的正确性，处理器性能也会受损。参见 3.11.“转换后备缓冲区（TLB)”的详细说明。

PCE **启用性能监测计数器(CR4 的第 8 位)。**置 1 则允许任何保护特权级程序执行 RDPMC 指令，置 0 则 RDPMC 指令只能在 0 级特权运行。

OSFXSR **操作系统对 FXSAVE 和 FXRSTOR 指令的支持(CR4 的第 9 位)。**置 1 则具有下列功能：(1)表明操作系统支持 FXSAVE 和 FXRSTOR 指令，(2)允许 FXSAVE 和 FXRSTOR 指令在保存和恢复 x87 FPU 和 MMX 寄存器内容的同时，也保存和恢复 XMM 和 MXCSR 寄存器的内容，(3)允许处理器执行除了 PAUSE、PREFETCHh、SFENCE、LFENCE、MFENCE、MOVNTI

和 CLFLUSH 之外的所有其它 SSE/SSE2/SSE3 指令。

置 0 则 FXSAVE 和 FXRSTOR 指令只保存和恢复 x87 FPU 和 MMX 寄存器的内容，不会保存和恢复 XMM 和 MXCSR 寄存器的内容。

另外，如果这个标志置 0，当处理器企图执行除了 PAUSE、PREFETCHh、SFENCE、LFENCE、MFENCE、MOVNTI 和 CLFLUSH 之外的任何其它 SSE/SSE2/SSE3 指令时，都会产生一个非法操作码异常(#UD)，操作系统和管理程序必须明确地设置这个标志。

注意

CPUID 特征标志 FXSR、SSE、SSE2 和 SSE3 分别表示在特定的 IA-32 处理器上，是否具有 FXSAVE/FXRESTOR 指令、SSE 扩展、SSE2 扩展以及 SSE3 扩展。OSFXSR 位则为操作系统启用这些特征提供了途径以及并指明了操作系统是否支持这些特征。

OSXMMEXCPT **操作系统支持未屏蔽 SIMD 浮点异常(CR4 的第 10 位)。**表明当 SIMD 浮点异常（#XF）产生时，操作系统支持调用异常处理程序来处理非屏蔽 SIMD 浮点异常。只有 SSE/SSE2/SSE3 的 SIMD 浮点指令会产生 SIMD 浮点异常。操作系统和管理程序必须明确设置这个标志。如果这个标志没有设置，当处理器检测到非屏蔽 SIMD 浮点异常时，将会产生一个非法操作码异常(#UD)。

2.5.1.CPUID 识别控制寄存器标志

控制寄存器 CR4 中的 VME、PVI、TSD、DE、PSE、PAE、MCE、PGE、PCE、OSFXSR 和 OSXMMEXCPT 都是模型相关的。所有的这些标志(除了 PCE 标志)在使用之前都可以通过 CPUID 指令来检查它们在当前处理器中是否已经实现。

2.6.系统指令汇总

系统指令用来处理系统级的功能，比如装载系统寄存器、管理高速缓存、管理中断或者设置调试寄存器。许多这种指令只能被操作系统或者管理程序执行(即运行在 0 级特权上的程序)。其它指令则是可以在任何特权级别上运行，因而应用程序也可以执行它们。

表 2-2 列出了系统指令，并指出它们能否被应用程序使用。这些指令在本手册第 2 卷的第 3 章和第四章“指令集参考”中有详细说明。

表 2-2 系统指令汇总

指令	指令描述	对应用程序有用？	禁止应用程序使用？
LLDT	装载LDT寄存器	否	是
SLDT	保存LDT寄存器	否	否
LGDT	装载GDT寄存器	否	是
SGDT	保存GDT寄存器	否	否
LTR	装载任务寄存器	否	是
STR	保存任务寄存器	否	否
LIDT	装载IDT寄存器	否	是
SIDT	保存IDT寄存器	否	否
MOV CRn	装载和保存控制寄存器	否	是
SMSW	保存MSW	是	否
LMSW	装载MSW	否	是
CLTS	清空CR0中的TS标志	否	是
ARPL	调整RPL	是[1]	否
LAR	装载访问特权	是	否
LSL	装载段界限	是	否
VERR	检验读	是	否
VERW	检验写	是	否
MOV DBn	装载和保存调试控制器	否	是
INVD	使Cache无效，不回写	否	是
WBINVD	使Cache无效，回写	否	是
INVLPG	使TLB项无效	否	是
HLT	停机	否	是
LOCK(前缀)	总线锁	是	否
RSM	从系统管理态返回	否	是
RDMSR[3]	读与模式相关寄存器	否	是

WRMSR[3]	写与模式相关寄存器	否	是
RDPMC[4]	读性能监测计数器	是	是[2]
RDTSC[3]	读时间戳计数器	是	是[2]

注意：

1.对CPL是1或2的应用程序有用。

2.由CPL是3的应用程序通过控制寄存器CR4中的TSD和PCE标志访问这些指令。

3.这些指令是在IA-32架构中的Pentium处理器引入的。

4.该指令是在IA-32架构中的Pentium Pro处理器和Pentium® MMX™处理器中引入的。

2.6.1 装载和保存系统寄存器

GDTR、LDTR、IDTR 和 TS 寄存器每个都有装载和保存指令用来从寄存器中装载或者保存到寄存器中去的指令：

LGDT(装载 GDTR 寄存器)　把 GDT 基地址和界限从内存装载到 GDTR 寄存器中。

SGDT(保存 GDTR 寄存器)　把 GDTR 寄存器中的 GDT 基地址和界限保存到内存中。

LIDT(装载 IDTR 寄存器)　把 IDT 基地址和界限从内存装载到 IDTR 寄存器中。

SIDT(保存 IDTR 寄存器)　把 IDTR 寄存器中的 IDT 基地址和界限保存到内存中。

LLDT(装载 LDT 寄存器)　从内存中装载 LDT 段选择子和段描述符到 LDTR 中。(段选择子操作数也可以位于通用寄存器。)

SLDT(保存 LDT 寄存器)　把 LDTR 寄存器中的 LDT 段选择子保存内存或者通用寄存器中。

LTR(装载任务寄存器)　把某个 TSS 的段选择子和段描述符从内存装载到任务寄存器中。(段选择子操作数也可以位于通用寄存器中)。

STR(保存任务寄存器)　把当前任务 TSS 的段选择子从任务寄存器保存到内存或者通用寄存器中。

LMSW(装载机器状态字)和 SMWS(保存机器状态字)指令操作控制寄存器 CR0 的 0 到 15 位。这两个指令是为兼容 16 位 Intel 286 处理器而提供的。为 32 位 IA-32 处理器写的程序不应该再使用这两个指令，而应该用 MOV 指令来访问 CR0。

CLTS(将 CR0 中的 TS 标志清零)指令为处理“设备不可使用”异常(#NM)而提供的，该异常在 TS 标志为 1 且当处理器试图执行浮点指令时出现。这个指令允许在保存 x87

FPU 场境以后清除 TS 标志，从而避免出现#NM 异常。参见 2.5.“控制寄存器”对这个标志的详细说明。

控制寄存器(CR0、CR1、CR2、CR3 和 CR4)都是用 MOV 指令来装载的。这个指令可以从通用寄存器中装载控制寄存器，也可以把控制寄存器保存到通用寄存器中。

2.6.2.检查访问特权

处理器提供了几条指令来检查段选择子和段描述符，以确定是否允许访问与它们相关联的段。这些指令复制处理器自动进行的某些访问特权和类型检查，从而允许操作系统和管理程序阻止某些异常的产生。

ARPL(调整 RPL)指令调整段选择子的 RPL(请求访问特权)来匹配那些提供该段选择子的程序/过程。参见 4.10.4.“检查调用者访问特权(ARPL 指令)”对这条指令功能和用法的详细介绍。

LAR(装载访问特权)指令检查某个段是否可以访问，并从段描述符中装载访问权限信息到通用寄存器中。然后，软件可以检查访问权限，看看段类型是否与使用意图兼容。参见 4.10.1.“检查访问权限(LAR 指令)”对这条指令功能和用法的详细说明。

LSL(装载段界限)指令检查某个段是否可以访问，并从段描述符中装载段界限到通用寄存器中。软件可以比较段界限和段内偏移，看看段内偏移是否在段内。参见 4.10.3.“检查指针偏移是否在界限之内(LSL 指令)”中对这条指令的功能和用法的详细介绍。

VERR(读检查)和 VERW(写检查)指令分别用来检查选定的段在某个 CPL 上是否可读或者可写。参见 4.10.2.“检查读/写权限(VERR 和 VERW 指令)”对这条指令功能和用法的说明。

2.6.3.装载和保存调试寄存器

处理器的内部调试设施是由一组 8 位调试寄存器(DR0 到 DR7)控制的。设置数据是用 MOV 指令装载或保存到/出这些寄存器中的。

2.6.4.使高速缓存和转换后备缓冲区（TLB）失效

处理器有几条指令用于使高速缓存和 TLB 失效。INVD 指令(使高速缓存失效，无回

写)使内部高速缓存中的所有数据和指令失效，并给外部高速缓存发送信号以指明它们也应该无效。

WBINVD 指令(使高速缓存失效，有回写)执行与 INVD 同样的功能，除此之外，还在失效之前，将内部内部高速缓存中修改过的行（line）回写到内存。使内部高速缓存失效后，WBINVD 给外部高速缓存发信号，让它们回写修改过的数据并使它们的内容失效。

INVLPG 指令(使 TLB 中某些区域失效)使 TLB 中指定的页失效（刷新该页）。

2.6.5.控制处理器

HLT 指令(停止处理器)停止处理器直至接收到一个启用中断(比如 NMI 或 SMI，正常情况下这些都是开启的)、调试异常、BINIT#信号、INIT#信号或 RESET#信号。处理器产生一个特殊的总线周期以表明进入停止模式。

硬件对这个信号的响应有好几个方面。前面板上的指示灯会打亮，产生一个记录诊断信息的 NMI 中断，调用复位初始化过程(注意 BINIT#引脚是在 Pentium Pro 处理器引入的)。如果停机过程中有非唤醒事件（比如 A20M#中断）未处理，它们将在唤醒停机事件处理之后的进行处理。

在修改内存操作时，使用 LOCK 前缀去调用加锁的读-修改-写操作(原子的)。这种机制用于多处理器系统中处理器之间进行可靠的通讯，具体描述如下：

在 Pentium 和早期的 IA-32 处理器中，LOCK 前缀会使处理器执行当前指令时产生一个 LOCK#信号，这总是引起显式总线锁定出现。

在 Pentium 4、Intel Xeon 和 P6 系列处理器中，加锁操作是由高速缓存锁或总线锁来处理。如果内存访问有高速缓存且只影响一个单独的高速缓存线，那么操作中就会调用高速缓存锁，而系统总线和系统内存中的实际内存区域不会被锁定。同时，这条总线上的其它 Pentium 4、Intel Xeon 或者 P6 系列处理器就回写所有的已修改数据并使它们的高速缓存失效，以保证系统内存的一致性。如果内存访问没有高速缓存且/或它跨越了高速缓存线的边界，那么这个处理器就会产生 LOCK#信号，并在锁定操作期间不会响应总线控制请求。

RSM 指令(从 SMM 返回)恢复处理器(从一个场境堆中)到系统管理态(SMM)中断之前的状态。

2.6.6.读取性能监测和时间戳计数器

RDPMC(读取性能监测计数)和 RDTSC(读取时间戳计数器)指令允许应用程序分别读取处理器的性能监测计数器和时间戳计数器。Pentium 4 和 Intel Xeon 处理器有 18 个 40 位的性能监测计数器，P6 系列处理器有 2 个 40 位的计数器。

这些计数器用来记录事件的发生及持续的时间。这些可以监视的事件都是模型相关的，可能包括：解码的指令条数、收到的中断个数、安装的高速缓存的个数。每个计数器都能用来监测一个不同的事件。使用系统指令 WRMSR 来在 45 个 ESCR 和 18 个 CCCRMSR (Pentium 4 和 Intel Xeon 处理器)、PerfEvtSel0 或者 PerfEvtSel1 MSR(P6 系列处理器)设置数值。RDPMC 指令用来从选定的计数器中装载当前计数值到 EDX：EAX 寄存器中。

时间戳计数器是一个模型相关的 64 位计数器，每次处理器复位后，它都置为 0。如果没有复位，当处理器运行在 3GHZ 时钟频率下时，计数器每年增加~9.5×10^{15}次。在这个时钟频率下运行 190 多年才会溢出。RDTSC 指令装载当前时间戳计数器的值到 EDX：EAX 寄存器中。

参见 15.8.“性能监测概况”和 15.7.“时间戳计数”对性能监测计数器和时间戳计数器的详细信息。

RDTSC 指令是随着 Pentium 引入 IA-32 架构的。RDPMC 指令是随着 Pentium Pro 和 Pentium MMX 处理器引入 IA-32 架构的。早期的 Pentium 有两个性能监测计数器，但是它们只能用 RDMSR 指令读取，并且只运行在 0 级特权上。

2.6.7.读写模型相关寄存器

RDMSR(读模型相关寄存器)和 WRMSR(写模型相关寄存器)分别允许对处理器的 64 位模型相关寄存器(MSR)进行读写。指定要被读写的 MSR 的值是放在 ECX 寄存器中的。

RDMSR 指令把指定的 MSR 的值读到 EDX：ECX 寄存器中；WRMSR 把 EDX：EAX 寄存器中的值写入指定的 MSR 中。RDMSR 和 WRMSR 指令是随着 Pentium 处理器引入到 IA-32 架构中的。

参见 9.4.“模型相关寄存器(MSR)”中对 MSR 的详细介绍。

第 3 章 保护模式内存管理

本章描述 IA-32 架构的保护模式内存管理设施，包括物理内存需求、分段机制和分页机制。关于处理器保护机制的描述参看第 4 章“保护”。关于实地址模式和虚拟 8086 模式内存寻址保护的描述参看第 16 章“8086 仿真”。

3.1. 内存管理概述

IA-32 架构的内存管理分为两个部分：分段和分页。分段提供了一种隔离每个进程或者任务代码、数据和栈模块的机制，保证多个进程或者任务能够在同一个处理器上运行而不会互相干扰。分页机制实现了传统请求调页的虚拟内存系统，在这种系统中，程序的执行环境块按需要被映射到物理内存中。分页机制同样可以用来隔离多个任务。在保护模式下，分段机制是必须的，分页机制则是可选的。**注意：没有一个标志位可用来关闭分段功能。**

对分页和分段机制进行不同的配置，可以分别支持简单的单任务系统、多任务系统或者使用共享内存的多处理器系统。

如图 3-1 所示，分段将处理器可寻址空间（即**线性地址空间**）分为较小的受保护的地址空间：**段**。段可以被用来装载一个程序的代码、数据或者堆栈，亦或装载系统的数据结构（如 TSS、LDT 等）。当处理器上运行多个进程时，可以为每个进程分配属于它自己的段集合。处理器会强制规定这些段的边界，以确保不会因为一个进程对属于另一个程序的段进行误写而互相干扰执行。分段机制也可以对段进行分类，确保操作限于某种类型的段。

系统中所有的段都包含在处理器的线性地址空间内。只有**逻辑地址**（也称为远指针）才能确定一个字节在一个特定段中的位置。逻辑地址由段选择子和偏移组成。段选择子是一个段的唯一标识。除此以外，它还是描述符表（比如全局描述符表 GDT）中的偏移，供访问一个叫做段描述符的数据结构之用。每个段有一个段描述符，段描述符描述段的大小、访问权限、段的优先权、段的类型以及段的第一个字节在线性地址空间的位置（也称为段基地址）等。通过将逻辑地址中的偏移部分加上段基地址就可以确定这个地址在段中的字节位置。段基地址加偏移就构成了处理器线性地址空间的

线性地址。

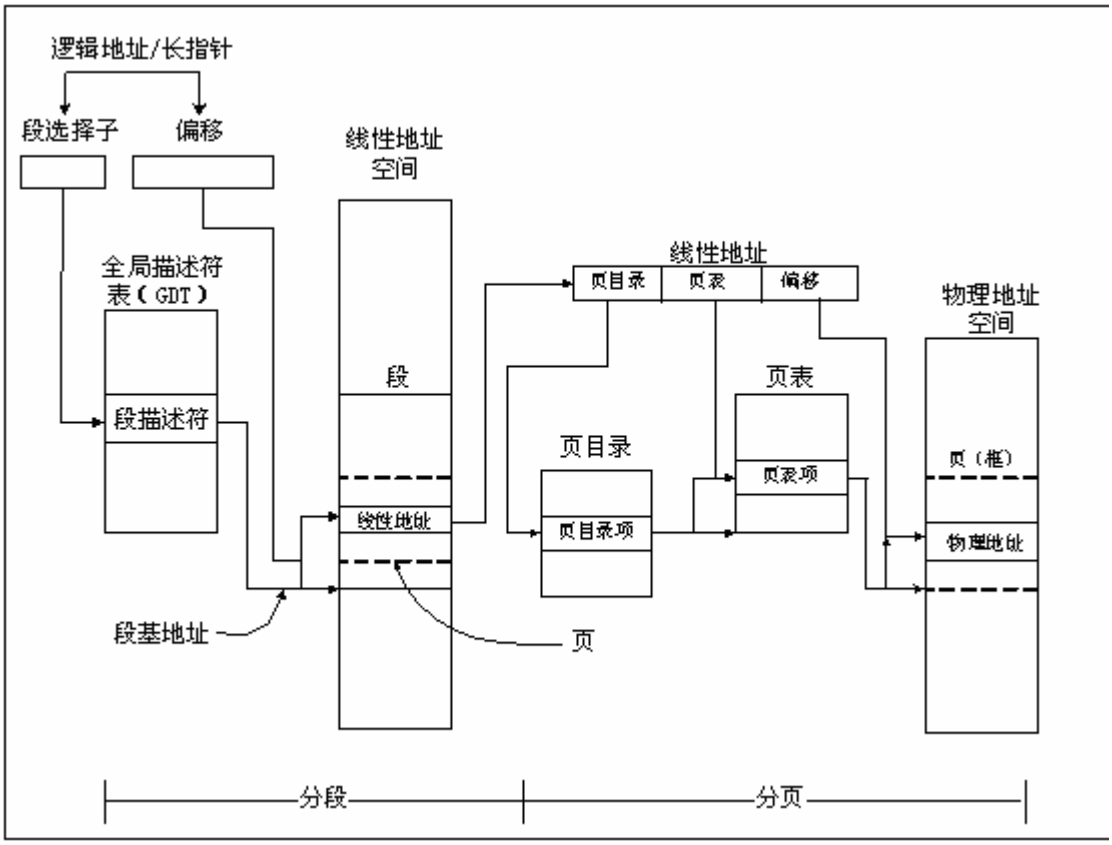

图 3-1 分段和分页

如果系统没有采用分页机制，线性地址空间就直接映射到物理地址空间。物理地址空间被定义为处理器能够在其地址总线上产生的地址范围。

在多任务计算系统中，通常会定义一个比实际物理内存空间大的多的线性地址空间，因此需要使用一些方法来“虚拟化”线性地址空间。线性地址空间的虚拟化由处理器的分页机制来完成。

分页机制支持虚拟内存环境，通过较小的物理内存（RAM 和 ROM）以及一些磁盘存储空间来模拟一个很大的线性地址空间。使用分页机制时，每个段被分成很多页（通常一个页的大小为 4KB），这些页或者在物理内存中，或者在磁盘上。操作系统会维护一张页目录表和一组页表来记住这些页。当一个进程/任务试图访问线性地址空间的一个地址时，处理器通过页目录表和页表将线性地址转换成物理地址，然后对相应的物理地址执行要求的操作（读或写）。

如果被访问的页不在当前的物理内存中，处理器会中断这个进程的执行（产生一个缺页异常）。之后，操作系统或者管理程序会从磁盘上读取这个页到内存中，接着执行这个程序。

一旦操作系统或者管理程序实现了分页机制，内存与磁盘之间的页交换对一个程序的正确执行来说是透明的，即使是为 16 位 Intel 处理器所写的程序，一旦运行在虚拟 8086 模式下，也可以被透明地分页。

3.2. 段的使用

IA-32 架构所支持的分段机制可被用来实现各种不同的系统设计。这些设计可以是平坦模型，即仅仅利用分段来保护程序。也可以是充分利用分段机制来实现一个健壮的操作系统，让多个程序安全可靠的运行。

以下给出几个例子来说明如何在一个系统中应用分段机制来改进内存管理的性能和可靠性。

3.2.1. 基本平坦模型

对一个系统而言，最简单的内存模型就是基本的“平坦（flat）模型”。在平坦模型中，操作系统和应用程序可以访问一个连续的、没有分段的地址空间。无论对系统设计者还是应用程序员，平坦模型在最大程度上隐藏了 IA-32 架构的分段机制。

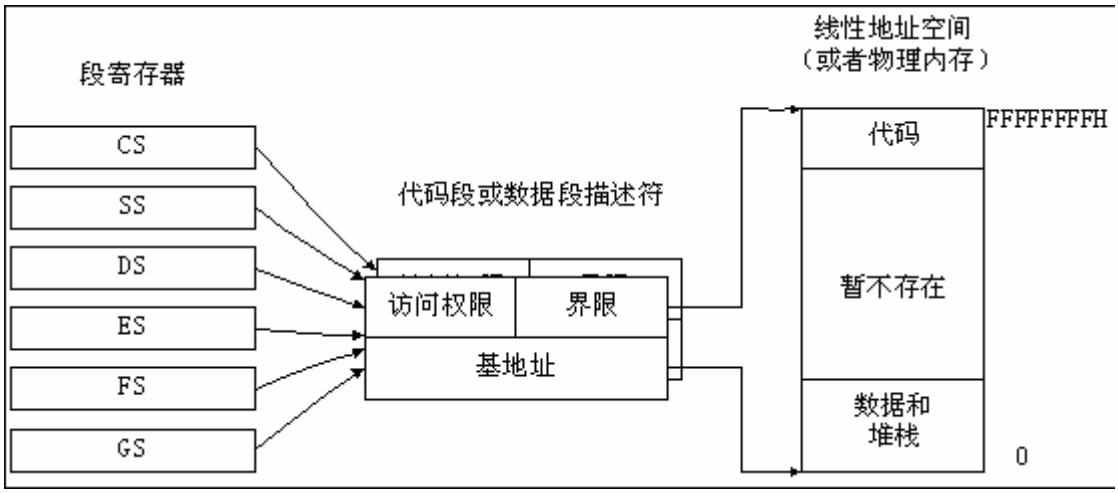

图 3-2 平坦模型

在 IA-32 架构中实现一个基本的平坦模型，至少要建立两个段描述符，一个指向代码段，一个指向数据段（具体请参考图 3-2）。这两个段都要被映射到整个线性地址空间，也就是说，这两个段描述符都是以地址 0 为基址，有同样的段界限 4GB。当段界限设置为 4GB 时，即使所访问的地址处并没有物理内存时，处理器也不会产生“超出

内存范围”异常。ROM（EPROM）的地址通常位于物理地址空间的高端，因为处理器从 ffff fff0H 处开始执行。RAM（DRAM）位于地址空间的低端，因为复位初始化后，数据段 DS 的初始基地址被置为 0。

3.2.2. 保护平坦模型

保护平坦（Protected Flat）模型与基本平坦模型类似，只是段界限被设定为在实际物理内存范围内（参看图 3-3）。如果试图访问实际内存范围以外的地址，会产生一个一般保护异常（#GP）。这个模型最小程度地利用了硬件的保护机制来防止一些程序错误。

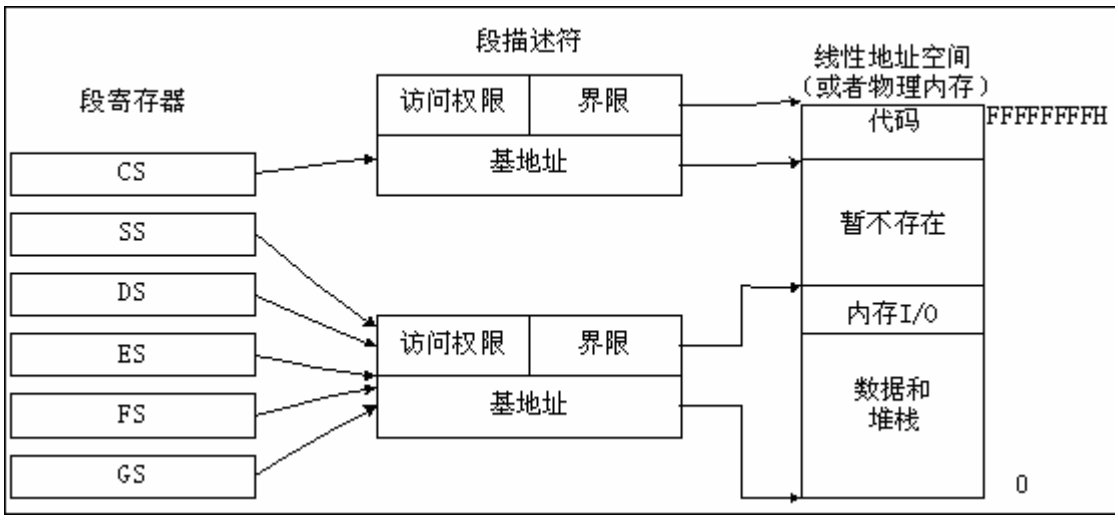

图 3-3 保护平坦模型

当然，也可以增加保护平坦模型的复杂性以提供更多的保护。比如，为了能在分页机制中隔离用户和管理代码和数据，需要定义 4 个段：特权级为 3 的用户代码段和数据段和特权级为 0 的管理代码段和数据段。一般来说，这些段都是互相重叠的，都从线性地址空间的地址 0 开始。这种平坦分段模型加上一个简单的分页结构就保护操作系统免受应用程序（干扰）。如果再为每一个任务或者进程各分配一个分页结构，也可以在应用程序之间起到保护作用。在几个常见的多任务操作系统中，采用了类似的设计。

3.2.3. 多段模型

多段模型（如图 3-4 所示）充分利用分段机制，提供对代码、数据结构、任务或

者程序的硬件强制保护。在这里，每个进程（或者任务）都分配有自己的段描述符和相应的段。段可以完全归它们所分配的程序独占，也可以供其它进程共享。对所有段的访问和对运行在系统中的每个程序执行环境的访问都受硬件控制。

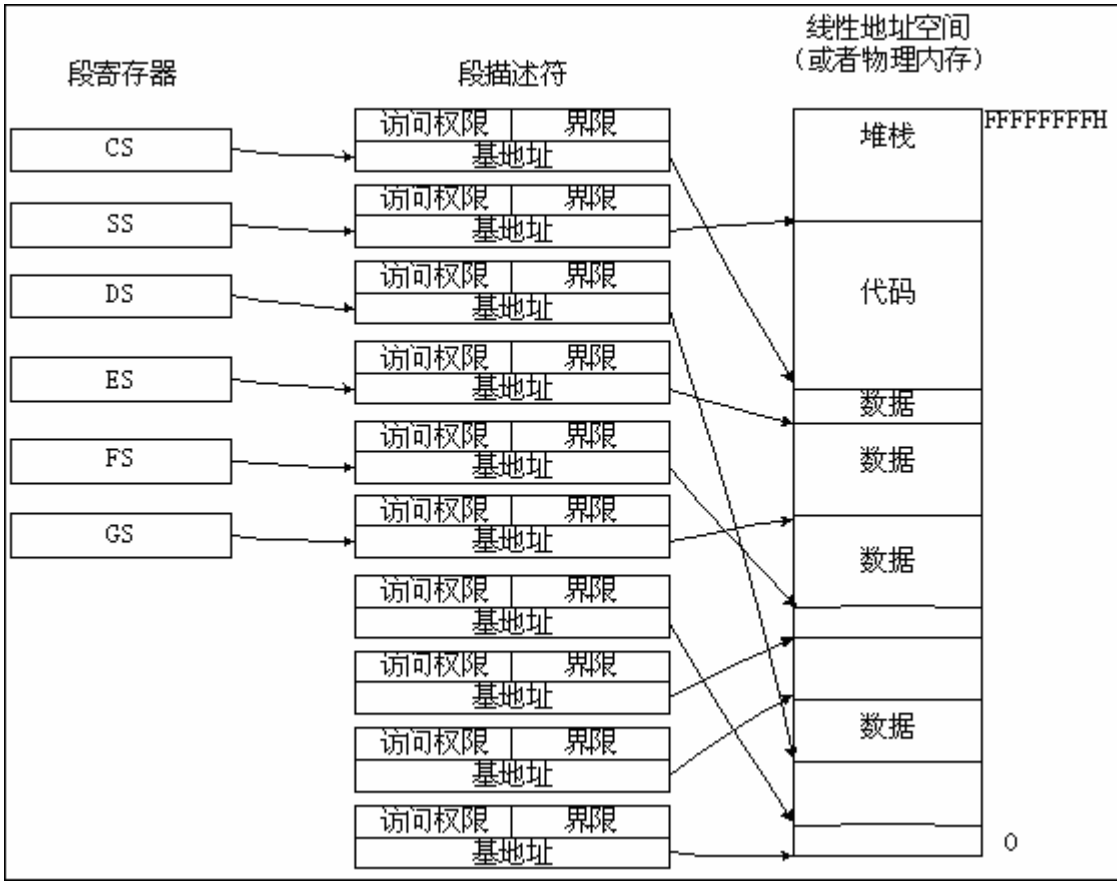

图 3-4 多段模型

访问检查机制不仅可以禁止对超过段界限的地址进行寻址，也可以禁止对特定段执行非法操作。比如，因为代码段被指定为只读的，所以用硬件来防止对代码段写入数据。段的访问权限信息也可以被用来建立保护级别。保护级别可以用来防止操作系统的例程被未授权的应用进程访问。

3.2.4. 分页与分段

分页机制可在图 3-2、3-3、3-4 所描述的任何一种分段模型中使用。处理器的分页机制把线性地址空间（段映射出来的）分成很多页（如图 3-1 所示），线性地址空间里的页再被映射到物理地址空间上的页。分页机制提供了一些页级保护设施，这些设施可以与段保护设施配合使用或者取代段的保护措施。例如，它可以强制以页为基础，实施读写保护。分页机制也可以提供两级即用户级和管理级的保护，这些保护也可以

以页为基础来实现。

3.3. 物理地址空间

在保护模式中，IA-32 架构正常情况下可以提供 4GB（2^{32} 字节）的物理地址空间，这是它的地址总线能够寻址的范围。这个地址空间是平坦的（未分段的），范围从 0 到 FFFFFFFFH。这个地址空间可以映射到读写内存、只读内存以及 I/O 内存。本章所描述的内存映射设施可以将物理内存分割成段或者页。

从 Pentium Pro 处理器开始，IA-32 架构支持的物理内存空间扩展至 2^{36} 字节（64GB），最大物理地址为 FFFFFFFFFH。下面方式的任何一种都可以启用这种扩展：

- 使用物理地址扩展（PAE）标记，它是控制寄存器 CR4 的第 5 位。
- 使用 36 位页尺寸扩展（PSE-36）特征（Pentium III处理器引入的）。

有关更多 36 位物理地址寻址的信息参看 3.8.“使用 PAE 分页机制进行 36 位物理地址寻址”和 3.9.“使用 PSE-36 分页机制进行 36 位物理地址寻址”。

3.4. 逻辑地址和线性地址

在保护模式的系统架构中，处理器分两步进行地址转换以得到物理地址：逻辑地址转换和线性地址空间分页。

即使最小程度的使用段机制，处理器地址空间内的每一个字节都是通过逻辑地址访问的。一个逻辑地址由一个 16 位的段选择子和一个 32 位的偏移组成（参考图 3-5）。段选择子确定字节位于哪个段，偏移确定字节在段中相对于段基地址的位置。

处理器将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内的 32 位的地址。与物理地址一样，线性地址是平坦的（不分段的），空间大小为 2^{32} 字节，从地址 0 到 FFFFFFFFH。线性地址空间包含了所有的段和为系统定义的各种系统表。

处理器通过如下几个步骤将逻辑地址转换为线性地址：

1．使用段选择子中的偏移，找到 GDT 或者 LDT 中相应的段描述符，得到段信息并把它读到处理器中。（仅当一个新的段选择子被读入段寄存器时才执行这一步。）

2．检查段描述符中的访问权限和段的地址范围，确保段是可访问的且偏移在段界限范围内。

3．将段描述符中的段基址与偏移相加构成线性地址。

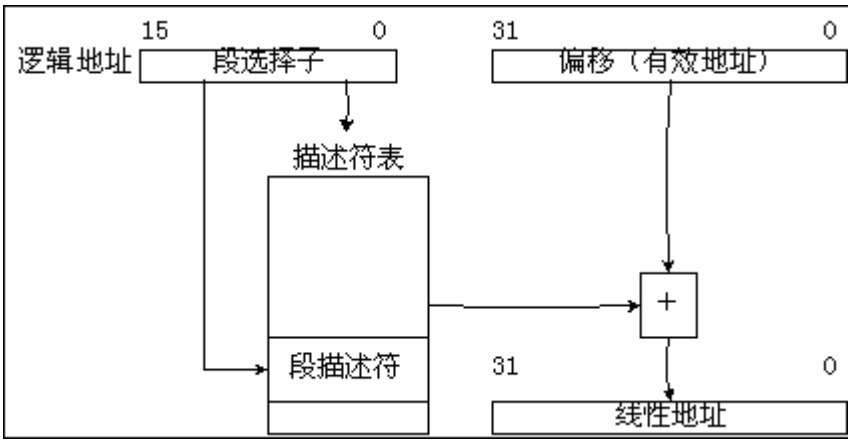

图 3-5 逻辑地址到线性地址的转换

如果没有使用分页，处理器直接将线性地址映射为物理地址（也就是说线性地址直接送到处理器的地址总线上）。如果在线性地址空间启用了分页机制，就需要再进行一次地址转换，将线性地址转换为物理地址。页变换的描述请参照 3.6.“分页（虚拟内存）概述”。

3.4.1.段选择子

段选择子是一个16 位的段的标识码（参照图 3-6）。它并不直接指向段，而是指向定义段的段描述符。段选择子包含以下项目：

索引 （第 3 位到第 15 位）。选择 GDT 或 LDT 中 8192 个描述符中的某一个。处理器将索引值乘以 8（段描述符的字节数），然后加上 GDT 或 LDT 的基地址（分别在 GDTR 或者 LDTR 寄存器中）。

表指示标记（TI） （第 2 位）。确定使用哪一个描述符表：将这个标记置 0，则表示用 GDT。将这个标记置 1，则表示用 LDT。

请求的特权级（RPL） （第 0 位和第 1 位）。确定选择子的特权级。特权级的范围是 0 到 3，0 为最高特权级。有关 RPL 与当前任务的 CPL 之间的关系以及段选择子所指的描述符的描述符特权级（DPL）的描述，参见 4.5.“特权级”。

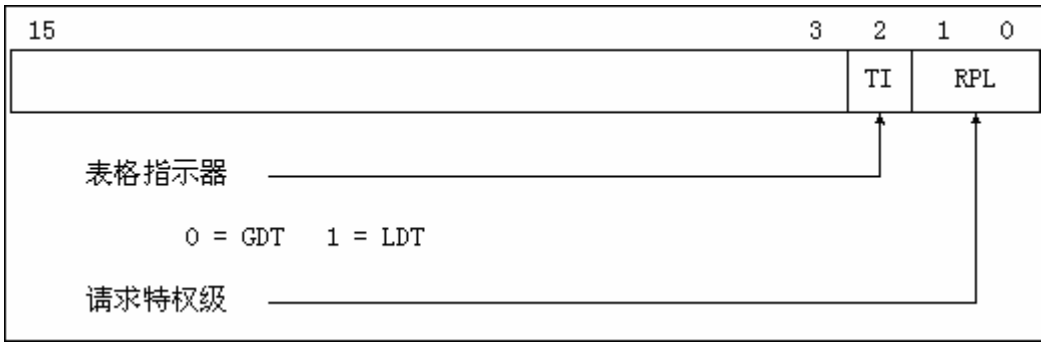

图 3-6 段选择子

GDT 的第一项是不用的。指向 GDT 第一项的段选择子（即选择子中的索引为 0，表指示标记 TI 置为 0）被视为“空段选择子”。当段寄存器（不是 CS 或 SS 寄存器）装载一个空选择子时，处理器并不产生异常。但是，当用装有空选择子的段寄存器来访问内存时，处理器则会产生异常。空选择子可以用来初始化未使用的段寄存器。对 CS 或者 SS 装载一个空选择子会产生一个一般保护异常（#GP）。

段选择子作为指针变量的一部分对应用程序而言是可见的，但是其值由连接编译器或者连接装载器程序赋予或者更改，而不是应用程序。

3.4.2. 段寄存器

为了减少地址转换的时间和编码复杂度，处理器提供了 6 个段寄存器来保存段选择子（参见图 3-7）。每个段寄存器都支持某个特定类型的内存寻址（代码、堆栈、数据等等）。实际上，对任何程序的执行而言，至少要将代码段寄存器（CS），数据段寄存器（DS）和堆栈段寄存器（SS）赋予有效的段选择子。此外，处理器还提供了另外 3 个数据段寄存器（ES、FS 和 GS）供当前执行程序（任务）增加数据段时使用。

段的段选择子必须被装载到某一个段寄存器中才可以让程序访问段。因此，尽管系统可以定义数千个段，但只有 6 个段是可以被直接使用的。其它段只有在程序执行期间把它们的段选择子装入这些段寄存器中才可以被使用。

可见部分	隐藏部分	
段选择子	基地址、界限、访问权限信息	CS
		SS
		DS
		ES
		FS
		GS

图 3-7 段寄存器

每个段寄存器都由“可见”部分和“不可见”部分组成。(有时，不可见部分也被称为“描述符高速缓存”或者“影子寄存器”)。当段选择子被装载到一个段寄存器的可见部分时，处理器也把段选择子指向的段描述符中的段基地址、段界限和访问权限等信息装载进来。段寄存器保存的信息（可见或不可见的）使得处理器在进行地址转换时，不需要花费额外的总线周期来从段描述符中获取段基地址和段界限。在允许多个处理器访问同一个描述符表的系统中，当描述符表被改变后，软件应该重新载入段寄存器。如果不这么做，那么，在内存驻留信息（its memory-resident version）发生变化后，使用的仍然是缓存在段寄存器中的旧的描述符信息。

有两种载入段寄存器的指令：

1. 直接载入指令，如 MOV、POP、LDS、LES、LSS、LGS 和 LFS。这些指令明确指定了相应的寄存器。

2. 隐含的载入指令，如远指针型的 CALL、JMP、RET 指令，SYSENTER 和 SYSEXIT 指令，还有 IRET、INT*n*、INTO 和 INT3 指令。伴随这些指令的操作，改变了 CS 寄存器（有时也包括其它段寄存器）的内容。

MOV 指令也可以用于将一个段寄存器的可见部分保存到一个通用寄存器中。

3.4.3. 段描述符

段描述符是 GDT 或 LDT 中的一个数据结构，它为处理器提供诸如段基地址、段大小、访问权限及状态等信息。段描述符主要是由编译器、连接器、装载器或者操作系统/管理程序设立的，而不是由应用程序产生的。图 3-8 说明了各类段描述符的一般格式。

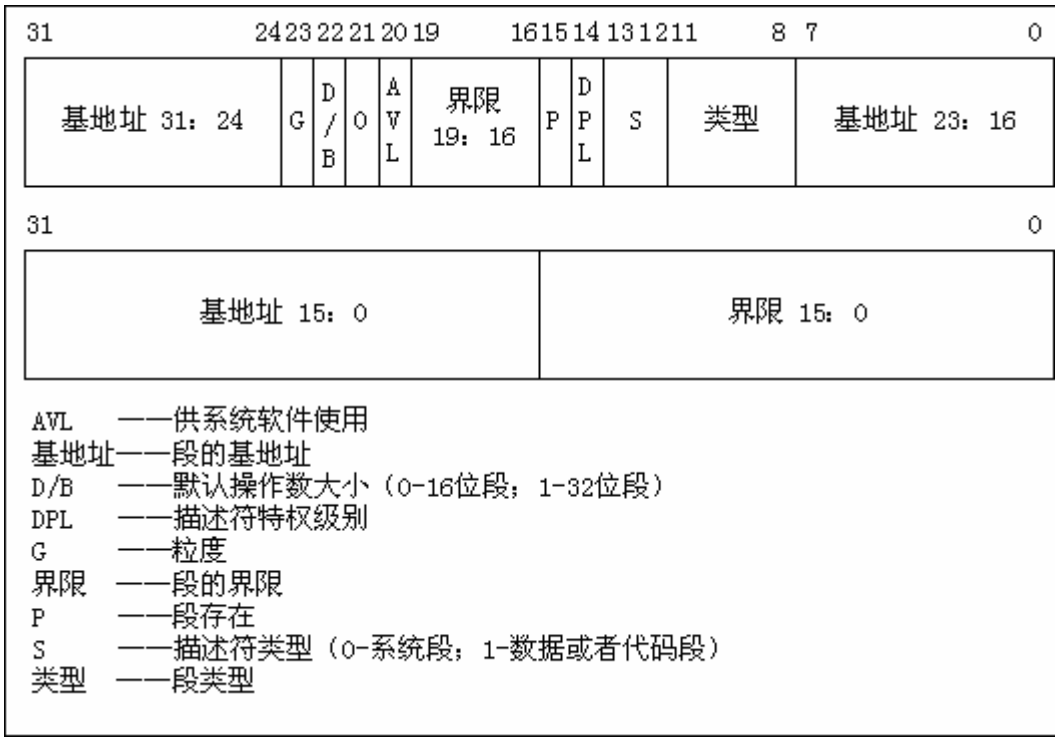

图 3-8 段描述符

段描述符中的标志和域如下：

段界限域 指定段的大小。处理器将两个段界限域合在一起组成一个 20 位的段界限值。根据 G 标志位（粒度）的不同设置，处理器按两种不同的方式解释段界限：

- 如果粒度标志位为 0，则段大小可以从 1 字节到 1M 字节，段长增量单位为字节。
- 如果粒度标志位为 1，则段大小可以从 4K 字节到 4G 字节，段长增量单位为 4K 字节。

根据段是“向上扩展段”还是“向下扩展段”，处理器对段界限有两种不同的处理方式。更多段类型的内容参见 3.4.3.1.“代码段和数据段描述符类型”。在向上扩展段中，逻辑地址偏移的范围从 0 到段界限。超过段界限的偏移会导致一般保护异常（#GP）。在向下扩展段，段界限的作用正好相反；偏移的范围从段界限到 FFFFFFFFH 或者 FFFFH，这由标志位 B 的值决定。小于段界限的偏移也会导致一般保护异常。减少向下扩展段的段界限将会在段地址空间的底部而不是顶部为该段分配新的内存空间。IA-32 架构中的栈总是向下增长

的，采用这种机制便于实现可扩展的栈

基地址域 确定段的第一个字节（字节 0）在 4GB 线性地址空间中的位置。处理器将 3 个基地址域组合在一起构成了一个 32 位地址值。段基地址应当是 16 字节边界对齐的。16 字节边界对齐不是必须的，但这种段边界的对齐能够使程序把代码和数据对其到 16 字节边界上而最大化其性能。

类型域 指明段或者门的类型，确定段的访问权限和增长方向。如何解释这个域，取决于该描述符是应用程序描述符（代码或数据）还是系统描述符。代码段、数据段和系统段对类型域有不同的编码（参见图 4-1）。3.4.3.1“代码和数据段描述符类型”对此进行了描述

S（描述符类型）标志 确定段描述符是系统描述符（S 标记为 0）或者代码、数据段描述符（S 标记为 1）。

DPL（描述符特权级）域 指明段的特权级。特权级从 0 到 3，0 为最高特权级。DPL 用来控制对段的访问。有关 RPL 与当前任务的 CPL 之间的关系以及段选择子所指的描述符的描述符特权级（DPL）的描述，参见 4.5.“特权级”。

P（段存在）标志 指明段当前是否在内存中（1 表示在内存中，0 表示不在）。当指向段描述符的段选择子被装进段寄存器时，如果这个标志为 0，处理器会产生段不存在异常（#NP）。内存管理软件可以通过这个标志，来控制在某个特定时间有哪些段是真正的被载入物理内存的。这是除分页之外的另一个虚拟内存控制机制。

图 3-9 演示了段存在标志置 0 时段描述符的格式。当这个标志置 0 时，操作系统或者管理软件就可以随意去使用标明为“可用”的地方（段描述符里）来存贮自己的数据，比如有关已消失段的所在位置的信息。

D/B（默认操作数大小/默认栈指针大小和/或上限）标志 根据段描述符所指的是可执行代码段、向下扩展的数据段还是堆栈段，这个标志有不同的功能。（对 32 位的代码和数据段，这个标志总是被置为 1，而 16 位的代码和数据段，这个标志总是被置为 0。）

- **可执行代码段**。这个标志被称为 D 标志，它指明段中指令的有效地址和操作符的默认位数。如果该标志为 1，则默认 32 位的地

址，32 位或者 8 位的操作符；若为 0，则默认 16 位的地址，16 位或者 8 位的操作符。指令前缀 66H 可以指定操作符的长度而不使用缺省长度。前缀 67H 可用来指定地址值的长度。

- **堆栈段（SS 寄存器所指的数据段）**。这个标志被称为 B（大的）标志，它为隐含的栈操作（如 push、pop 和 call）确定栈指针的大小。如果该标志为 1，则使用 32 位的栈指针，栈指针放在 32 位的 ESP 寄存器中；若该标志为 0，则使用 16 位的栈指针，栈指针存放在 SP 寄存器中。如果堆栈段为一个向下扩展的数据段（见下一段的说明），则 B 标志还确定堆栈段的地址上界。
- **向下扩展的数据段。**这个标志称为 B 标志，确定段的地址上界。如果该标志为 1，则段地址上界为 FFFFFFFFH（4GB）；若该标志为 0，则段地址上界为 FFFFH（64KB）。

G（粒度）标志 确定段界限扩展的增量。当 G 标志为 0，则段界限以字节为单位扩展；G 标志为 1，则段界限以 4KB 为单位扩展。（这个标志不影响段基址的粒度，段基址的粒度永远是字节。）如果 G 标志为 1，那么当检测偏移是否超越段界限时，不用测试偏移的低 12 位。例如，如果 G 标志为 1，0 段界限意味着有效偏移为从 0 到 4095。

可用及保留位 段描述符的第二个双字的第 20 位可以被系统软件使用，第 21 位被保留，并且应该设置为 0。

图 3-9 段存在标志为 0 时的段描述符

3.4.3.1.代码段和数据段描述符类型

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）。

表 3-1 代码段和数据段类型

类型域					描述符类型	说　　明
段类型（十进制数）	11	10 E	9 W	8 A		
0	0	0	0	0	数据	只读
1	0	0	0	1	数据	只读，已被访问
2	0	0	1	0	数据	读/写
3	0	0	1	1	数据	只写，可访问
4	0	1	0	0	数据	只读，向下扩展
5	0	1	0	1	数据	只读，向下扩展，已被访问
6	0	1	1	0	数据	读/写，向下扩展
7	0	1	1	1	数据	读/写，向下扩展，已被访问
		C	**R**	**A**		
8	1	0	0	0	代码	只执行
9	1	0	0	1	代码	只执行，已被访问
10	1	0	1	0	代码	执行/读
11	1	0	1	1	代码	执行/读，已被访问
12	1	1	0	0	代码	只执行，一致性的
13	1	1	0	1	代码	只执行，一致性的，已被访问
14	1	1	1	0	代码	执行/读，一致性的，
15	1	1	1	1	代码	执行/读，一致性的，已被访问

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）。参见表 3-1 对代码和数据段描述符类型域的解码描述。数据段可以是只读或者可读写的段，这取决于“是否可写”标志。

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。

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。

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）。

根据可读位的设置，代码段可以为“只执行”或者“可执行可读”。当有常量或者其它静态数据与指令代码一起在 ROM 中时，必须使用“可执行可读”的段。要从代码段读取数据，可以通过带有 CS 前缀的指令或者将代码段选择子置入数据段寄存器（DS、ES、FS 或者 GS）。在保护模式中，代码段是不可写的。

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）（更多关于一致和非一致性代码段的信息参看 4.8.1.“直接调用或跳转到代码段”）。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。

注意：

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。

如果 GDT 或者 LDT 中的段描述符被放置在 ROM 中，当程序或者处理器试图更改在 ROM 中的段描述符时，处理器将进入一个死循环。为了防止此类问题的发生，可以将所有放置在 ROM 中的段描述符的访问位置位。同时，删除所有操作系统代码中试图更改放置在 ROM 中的段描述符的代码。

3.5. 系统描述符类型

当段描述符的 S 标志（描述符类型）为 0，则描述符为系统描述符。处理器可以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。
- 任务门描述符。

这些描述符又可以分为两类：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）。门描述符它们自身就是“门”，它们或者持有指向放置在代码段中的过程入口点的指针，或者持有 TSS（任务门）的段选择子。表 3-2 列举了系统段描述符和门描述符类型域的组合说明。

表 3-2 系统段和门描述符类型

类型域					说　　明
类型 （十进制数）	11	10	9	8	
0	0	0	0	0	保留
1	0	0	0	1	16 位 TSS（可用）
2	0	0	1	0	LDT
3	0	0	1	1	16 位 TSS（忙）
4	0	1	0	0	16 位调用门
5	0	1	0	1	任务门
6	0	1	1	0	16 位中断门
7	0	1	1	1	16 位陷阱门
8	1	0	0	0	保留
9	1	0	0	1	32 位 TSS（可用）
10	1	0	1	0	保留
11	1	0	1	1	32 位 TSS（忙）
12	1	1	0	0	32 位调用门
13	1	1	0	1	保留
14	1	1	1	0	32 位中断门
15	1	1	1	1	32 位陷阱门

有关系统段描述符的更多描述参看 3.5.1.“段描述符表”和 6.2.2.“TSS 描述符”。有关门描述符的信息参看 4.8.3.“门描述符”、5.11.“IDT 描述符”和 6.2.4.“任务门描述符”。

3.5.1. 段描述符表

段描述符表是一个段描述符的数组（参看图 3-10）。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）

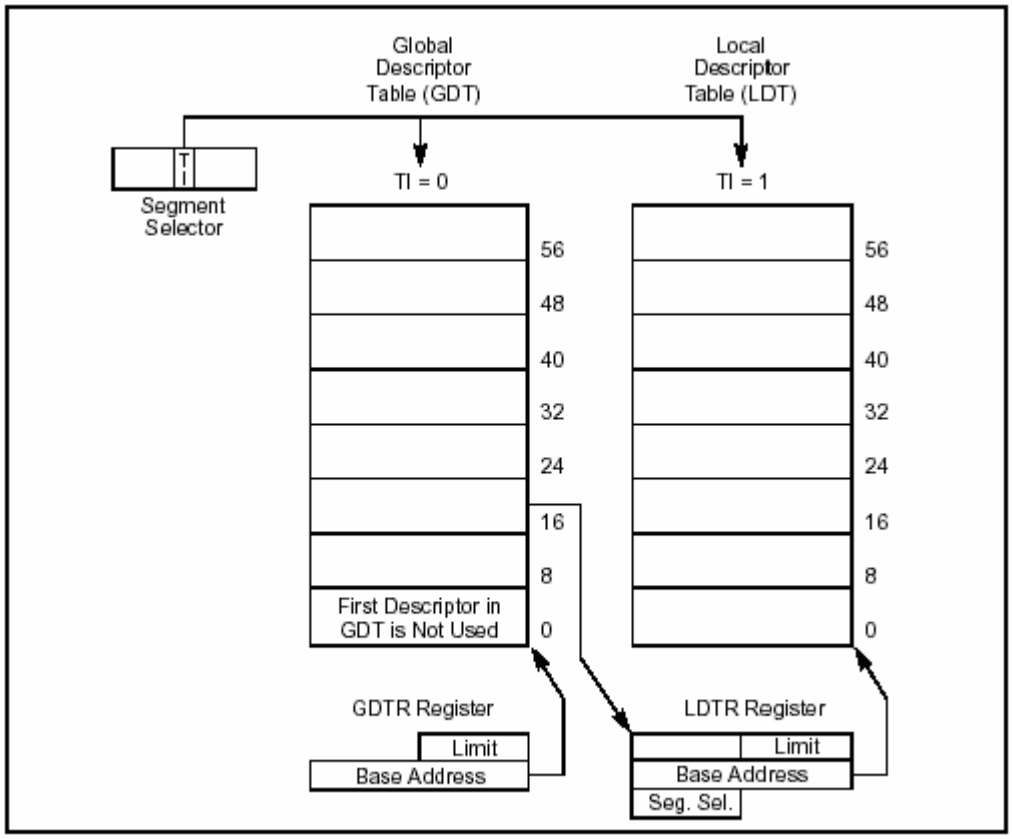

Figure 3-10. Global and Local Descriptor Tables

图 3-10 全局和局部描述符表

系统必须定义一个 GDT，以备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。

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（参见 2.4.“内存管理寄存器”）。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）。

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。

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。LDT 的段描述符可以位于 GDT 中的任何地方。关于 LDT 段描述符类型的信息参见 3.5.“系统描述符类型”。

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（参见 2.4.“内存管理寄存器”）。

当保存 GDTR 寄存器时（使用 SGDT 指令），一个 48 位的伪描述符被存入内存中（参看图 3-11）。为了避免在用户态下（特权级为 3）发生对齐检查错误，伪描述符应该放在一个奇字地址上（即，该地址对 4 取模的结果为 2）。这样就让处理器先保存一个对齐的字，后面再跟着一个对齐的双字。用户态下的程序通常并不保存伪描述符，但是通过这种方式对齐伪描述符可以避免发生对齐检查故障。在使用 SIDT 指令保存 IDTR 寄存器时，也应该使用同样的对齐方法。当保存 LDTR 或者任务寄存器（分别使用 SLTR 和 STR 指令)时,伪描述符应该被放在一个双字地址上(即该地址对 4 取模的结果为 0)。

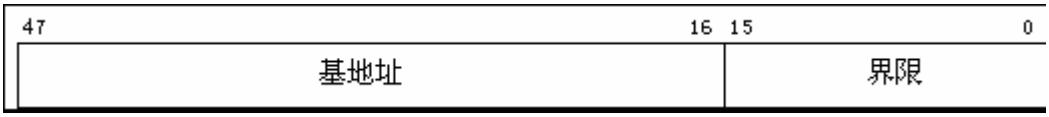

图 3-11 伪描述符格式

3.6.分页（虚拟内存）概述

当系统运行在保护模式下时,IA-32 架构允许将线性地址直接映射到一个大的物理空间（比如 4GB 的 RAM）或者间接的（使用分页）映射到一个较小的物理内存和磁盘存储空间中。后一种映射线性地址空间的方法通常被称作虚拟内存或者请求调页虚拟内存。

当使用分页时，处理器将线性地址空间划分成固定尺寸的页（4KB、2MB 或者 4MB），这些页可以被映射到物理内存或者磁盘存储空间。当一个程序（或者任务）引用内存中的一个逻辑地址时，处理器首先将它转换为线性地址，然后使用分页机制将线性地址转换为相应的物理地址。

如果包含线性地址的页目前不在内存中，处理器产生一个页故障异常（#PF）。页故障异常处理程序的典型的做法就是指导操作系统或者管理程序将相应的页从磁盘上调入内存（在这个过程当中可能有另外一个页被调出内存存放到磁盘上）。页被调入内存后，就从异常处理程序返回，重新执行导致页故障异常的指令。处理器用来映射线性地址到物理地址的信息和产生页故障异常（如有必要）的信息存放在页目录表和页表中，页目录表和页表都存放在内存中。

分页不同于分段，分页使用固定大小的页。与段不同，页有固定的尺寸，段通常与它所保存的代码或者数据结构的大小一样。如果只使用分段进行地址转换，则一个数据结构必须全部保存在物理内存中。如果启用了分页，一个数据结构可以部分在内存中，部分在磁盘中。

为了减少地址转换所使用的总线周期，最近被访问过的页目录表项和页表项都被高速缓存在一个叫做转换后备缓冲区（translation lookaside buffers，TLB）的设备中。TLB 可以满足多数的读当前页目录表和页表的请求而不使用总线周期。仅当所访问的页表项不在 TLB 中时，才需要额外的总线周期，而这种情景通常在访问一个很久不曾访问的页的时候才发生。关于 TLB 的更多内容参看 3.11.“转换后备缓冲区(TLB)”。

3.6.1.分页选项

分页由处理器的控制寄存器的 3 个标志来控制：

- **PG（分页）标志。**CR0 寄存器的第 31 位（从 Intel386 处理器开始的所有 Intel 处理器都有这个标志）。
- **PSE（页尺寸扩展）标志。**CR4 寄存器的第 4 位（Pentium 和 Pentium Pro 处理器引入的）。
- **PAE（物理地址扩展）标志。**CR4 寄存器的第 5 位（Pentium Pro 处理器引入的）。

PG 标志启用页转换机制。通常操作系统或者管理程序在处理器初始化的时候设置这个标志。如果处理器的页转换机制被用来实现请求调页虚拟存储系统，或者操作系统可以在虚拟 8086 模式下运行多个进程（或者任务），那么 PG 标志必须被设置。

PSE 标志启用更大尺寸的页：4MB 的页或者 2MB 的页（当 PAE 标志置 1 时）。当 PSE 标志被清零的时候，则使用通常的 4KB 页。更多有关 PSE 标志的使用信息参看 3.7.2.“线性地址转换（4MB 页）”、3.8.2.“开启 PAE 的线性地址转换（2MB 或者 4MB 的页）”

和 3.9.“使用 PSE-36 分页机制的 36 位物理寻址”。

PAE 标志提供了将内存地址扩展到 36 位的一种方法。仅当开启分页机制后才可以使用物理地址扩展。它依赖和页目录表、页表一起使用的页目录指针表来寻址超过 FFFFFFFFH 的地址。更多关于使用 PAE 来进行物理地址扩展的信息参看 3.8.“使用 PAE 分页机制的 36 位物理寻址”。

36 位页尺寸扩展（PSE-36）这个特征提供了一种扩展 36 位物理寻址的替代方法。这种分页机制使用页尺寸扩展模式（用 PSE 标志启用），并且更改页目录表项来寻址物理地址在 FFFFFFFFH 以上的内存。PSE-36 特征标志（当用源操作数 1 来执行 CPUID 指令时的 EDX 寄存器的第 17 位）指明了是否可用这种寻址机制。更多关于 PSE-36 物理地址扩展和页尺寸扩展机制的信息参看 3.9.“使用 PSE-36 分页机制进行 36 位物理寻址”。

3.6.2. 页表和页目录表

当启用分页机制时，处理器用来进行线性地址到物理地址转换的信息包含在下列 4 个数据结构中：

- 页目录表——一个由 32 位页目录表项（PDE）组成的数组，放置在一个 4KB 的页中。一张页目录表最多包含 1024 个页目录表项。
- 页表——一个由 32 位页表项（PTE）组成的数组，它存放于一个 4KB 的页中。一张页表最多包含 1024 个页表项。（2MB 或者 4MB 的页不使用页表，它们直接从一个或者多个页目录表项映射出来。）
- 页——一个 4KB、2MB 或者 4MB 的平坦地址空间。
- 页目录指针表——由 4 个 64 位的项组成的数组，每一项都指向一个页目录表。仅当启用物理地址扩展时才使用这个数据结构（参看 3.8.“物理地址扩展”）。

这些表可以用来在使用常规的 32 位物理寻址时访问 4KB 或者 4MB 的页，也可以用来在使用 36 位扩展物理寻址时访问 4KB、2MB 或者 4MB 的页。表 3-3 显示了分页控制标志与 PSE-36 的 CPUID 指令的各种设置下，页尺寸和物理地址尺寸的情况。每个页目录表项都包含了一个 PS（页尺寸）标志，这个标志指明该页目录表项是指向一张页表（其每个表项指向一个 4KB 的页）（PS 置为 0），还是直接指向一个 4MB（PSE 和 PS 置为 1）或者 2MB 的页（PAE 和 PS 为 1）。

3.7.使用 32 位物理寻址的页变换

本节描述使用 32 位物理地址的 IA-32 架构页变换机制，最大物理地址空间为 4GB。3.8.“使用 PAE 分页机制的 36 位物理寻址”和 3.9.“使用 PSE-36 分页机制的 36 位物理寻址”描述支持 36 位物理地址的页变换机制扩展，可支持最大 64GB 物理地址空间。

表 3-3 页尺寸和物理地址尺寸

Table 3-3. Page Sizes and Physical Address Sizes

PG Flag, CR0	PAE Flag, CR4	PSE Flag, CR4	PS Flag, PDE	Page Size	Physical Address Size
0	X	X	X	—	Paging Disabled
1	0	0	X	4 KBytes	32 Bits
1	0	1	0	4 KBytes	32 Bits
1	0	1	1	4 MBytes	32 Bits
1	1	X	0	4 KBytes	36 Bits
1	1	X	1	2 MBytes	36 Bits

3.7.1.线性地址转换（4KB 页）

图 3-12 展示了映射线性地址到 4KB 的页时的页目录表和页表的层次结构。页目录表项指向页表，页表项指向物理内存中的页。这种分页的方法可以用来寻址 2^{20} 张页，覆盖 2^{32} 个字节（4GB）的线性地址空间。

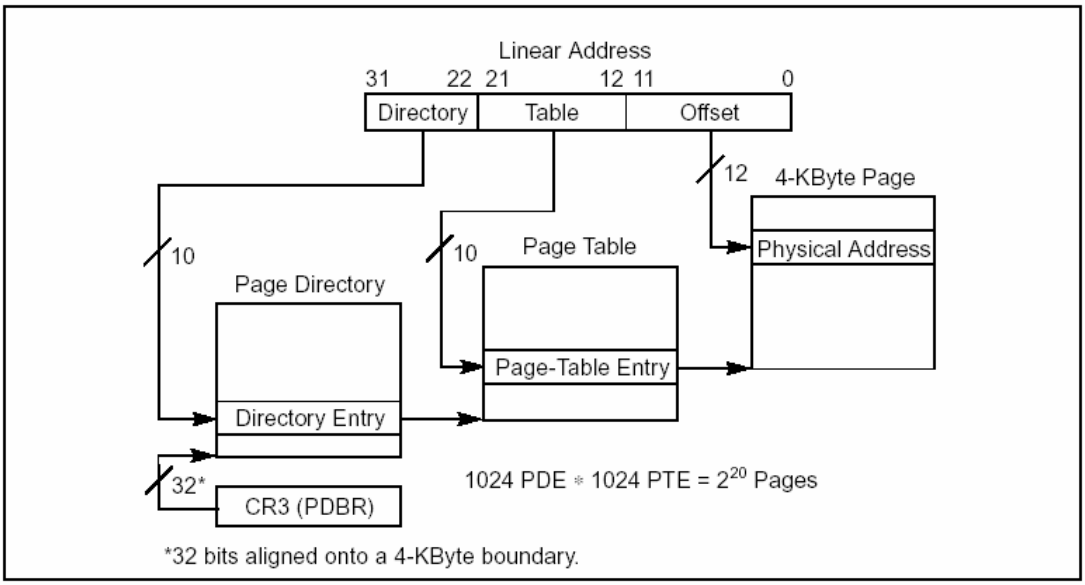

Figure 3-12. Linear Address Translation (4-KByte Pages)

图 3-12 线性地址转换（4KB 页）

为了选择不同的表的项，线性地址被分为 3 个部分:

- 页目录表项——第 22 位到第 31 位,作为表项在页目录表中的偏移。该表项提供所指页表的物理基地址。
- 页表项——第 12 位到第 21 位，作为表项在所选页表中的偏移。该表项提供所选物理内存页的物理基地址。
- 页偏移——第 0 位到第 11 位，提供地址在所选物理页中的偏移。

内存管理软件可以让所有的进程和任务使用一个页目录表，也可以使每个任务使用一个页目录表，或者两种方法联合使用。

3.7.2.线性地址转换（4MB 页）

图 3-13 显示如何使用页目录表来映射线性地址到 4MB 页。页目录表的项指向物理内存中的 4MB 页。这种分页方法可以将多达 1024 个页映射到 4GB 的线性地址空间。

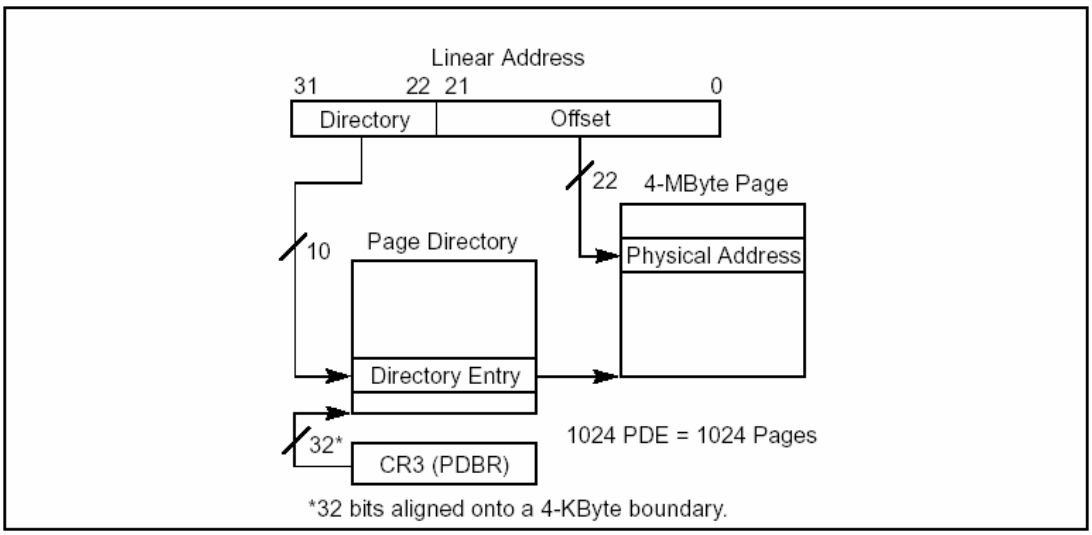

Figure 3-13. Linear Address Translation (4-MByte Pages)

图 3-13 线性地址转换（4MB 页）

通过设置控制寄存器 CR4 中的 PSE 标志和页目录表项中的页尺寸（PS）标志（参照图 3-14）来启用 4MB 页。这两个标志都置 1 时，则线性地址被分为两个部分：

- 页目录表项——第 22 位到第 31 位，提供目录项在页目录表中的偏移。目录项提供 4MB 页的物理基地址。
- 页偏移——第 0 位到第 21 位，提供物理地址在页中的偏移。

注意

（仅针对 Pentium 处理器。）当启用或者禁用大的页尺寸时，在设置或者清零控制寄存器 CR4 的 PSE 标志以后，TLB 必须被刷新。否则可能由于处理器使用了存在 TLB 中的过期的页转换信息而进行了错误的页转换。关于如何刷新(使之失效)TLB 参看 10.9.“使 TLB 失效”。

3.7.3.混合使用 4KB 和 4MB 页

当 CR4 中的 PSE 标志置 1 时，则可以通过同一个页目录表来访问 4MB 页和 4KB 页的页表。如果 PSE 标志被清零，就只能访问 4KB 页的页表（无论页目录表项中 PS 标志如何设置）。

混用 4KB 页和 4MB 页的典型例子是将操作系统或者管理程序内核放在大尺寸的页中来减少 TLB 命中率不高，从而整体上提高系统性能。

处理器在不同的 TLB 中维护 4MB 页项和 4KB 页项。因此，将频繁使用的代码，比如内核，放在大尺寸的页中，从而为应用进程和任务留出了 4KB 页的 TLB 项。

3.7.4.内存别名

IA-32 架构可以使用内存别名，方法是将两个页目录表项指向同一个页表。以这种方式实现内存别名的软件必须处理好页目录表项和页表项中“访问位”和“脏位”的一致性。允许两个页目录表项的访问位和脏位不一致，将会导致处理器死锁。

3.7.5.页目录表基地址

当前页目录表的物理地址存放在 CR3 寄存器中（也称为页目录基址寄存器 PDBR）。（更多关于 PDBR 寄存器的信息参看图 2-5 及 2.5.“控制寄存器”。）如果启用了分页，装载 PDBR 必须作为处理器初始化过程的一部分（在启用分页之前）。之后，可以通过使用 MOV 指令装载一个新值到 CR3 来显式地改变 PDBR 的值，或者在任务切换时隐式地改变它。(关于如何为任务设置 CR3 参看 6.2.1.“任务状态段（TSS)”。)

在 PDBR 中没有为页目录表而设的存在标志。当一个任务被挂起时，与之相关的页目录表也许不在内存（被请求调出物理内存)。但是操作系统要确保，在一个任务被调

度运行之前，该任务的 TSS 中的 PDBR 映像所指的页目录表必须已经在内存中。只要任务还是处于活动状态，页目录表就必须保留在内存中。

3.7.6.页目录表项和页表项

图 3-14 显示了使用 4KB 页和 32 位物理地址时，页目录表项和页表项的格式。图 3-15 显示了使用 4MB 页和 32 位物理地址时页目录表项的格式。图 3-14 和图 3-15 表项中的标志和域的功能阐述如下：

页基地址，第 12 位到第 32 位

（页表项，指向 4KB 页。）确定一个 4KB 页的第一个字节的物理地址。这个域被解释为物理地址的高 20 位，强迫页 4KB 对齐。

（页目录表项，指向 4KB 页表。）确定页表的第一个字节的物理地址。这个域被解释为物理地址的高 20 位，强迫页表 4KB 对齐。

（页目录表项，指向 4MB 页。）确定一个 4MB 页的第一个字节的物理地址。在这种情况下，只用了这个域的第 22 位到第 31 位（从 Pentium Ⅱ开始的 IA-32 架构处理器把第 12 位到第 21 位保留并置 0）。这些基地址位被解释为物理地址的高 10 位，强迫 4MB 页是 4MB 对齐的。

存在（P）标志，第 0 位

该标志表明表项所指的页或者页表当前是否在内存中。该标志置位时，则页在物理内存中，将执行地址转换。该标志清零时，表示页不在内存中，如果处理器试图访问该页，将产生一个页故障异常（#PF）。

处理器并不置位或者清零该位；而是由操作系统或者管理程序来维护该标志的状态。

如果处理器产生一个页故障异常，操作系统一般情况下必须执行如下操作：

1．将页从磁盘复制到内存中。

2．将页地址装入页表项或者页目录表项中并设置它的存在标志。其它标志位，比如脏位和访问位，也必须同时设置。

3．使 TLB 中的当前页表项失效（有关 TLB 的信息以及如何使它们失效参考 3.7.“转换后备缓冲区（TLB）”）。

4．从缺页异常处理程序返回，重新执行被中断的进程或任务。

读/写（R/W）标志，第 1 位

该标志确定对一个页或者一组页（当它是一个指向页表的页目录表项时）的读写权限。当这个标志被清零时，该页是只读的；当这个标志被置位，该页是可读可写的。

该标志与 U/S 标志和 CR0 寄存器中的 WP 标志相互影响。有关使用这些标志的详细讨论参看 4.11.“页级保护”和表 4-2。

用户/管理（U/S）标志，第 2 位

该标志确定一个页或者一组页（当它是一个指向页表的页目录表项时）是用户特权还是管理特权。当这个标志被清零，则页被赋予管理特权；该标志置位时，则页被赋予用户特权。这个标志与 R/W 标志和 CR0 寄存器中的 WP 标志相互影响。有关使用这些标志的详细讨论参看 4.11.“页级保护”和表 4－2。

页级直写（PWT）标记，第 3 位

控制单个页（页表）的直写或者回写高速缓存策略。当 PWT 标志被置位时，启用页表的直写高速缓存机制；当 PWT 标志被清零时，回写高速缓存与页或者页表关联；当 CR0 寄存器中的 CD（高速缓存禁用）标志被置位时，处理器忽略这个标志。有关使用这个标志的更多信息参看 10.5.“高速缓存控制”。关于控制寄存器 CR3 中与 PWT 标志共同起作用的标志的描述参看 2.5.“控制寄存器”。

页级高速缓存禁用（PCD）标志，第 4 位

控制单个页或者页表的高速缓存。当该标志被置位时，相关页或者页表的高速缓存被禁止；当该位被清零时，相关页或页表可以被高速缓存。这个标志可以用来禁止高速缓存包含内存映射 I/O 端口的页或者即使被高速缓存，也不能对性能有提高的页。当 CR0 寄存器中的 CD（高速缓存禁用）标志被置位时，处理器将忽略这个标志。更多关于这个标志的使用信息参看第 10 章“内存缓存控制”。有关结合 CR3 寄存器中的 PCD 标志使用的描述参看 2.5.“控制寄存器”。

访问（A）标志，第 5 位

指明页或页表是否曾经被访问过。内存管理软件通常会在页或者页表被载入内存时，清零该位。当页或者页表第一次被访问以后，处理器会置位该标志。

这个标志是个“粘性的”标志，就是说一旦被设置，处理器不会隐

式地给它清零。只有软件能清零该位。内存管理软件使用访问位和脏位来调度页或者页表进出物理内存。

注意：通过处理器来设置该位，有可能会也有可能不会曝露出处理器的自修改代码的检测逻辑（Self-Modifying Code detection logic），如果处理器是从与页表结构相同的内存地址执行代码的话，设置该位有可能会也有可能不会导致立即改变代码的执行。

脏（D）位，第 6 位

指明页是否曾经被写入过（在指向页表的页目录表项中，不使用该标志）。通常，内存管理软件在页刚被载入内存时，将该标志清零。当页的第一次写操作完成后，处理器置位该标志。

这个标志具有“粘性”，就是说，一旦被设置，处理器不会隐式地对它清零。只有软件可以对它清零。内存管理软件使用访问位和脏位来调度页或者页表进出物理内存。

注意：通过处理器来设置该位，有可能会也可能不会曝露出处理器的自修改代码的检测逻辑（Self-Modifying Code detection logic），如果处理器是从与页表结构相同的内存地址执行代码的话，设置该位有可能会也有可能不会导致立即改变代码的执行。

页尺寸（PS）标志，第 7 位

确定页的尺寸。该标志仅被用于页目录表项。当该标志被清零时，页尺寸为 4KB，页目录表项指向一个页表。当该标志被置位时，页的尺寸为 32 位寻址的 4MB（当扩展物理寻址启用时，页的尺寸为 2MB），页目录表项指向一个页。如果页目录表项指向一个页表，所有与那个页表相关的页都是 4KB。

页属性表索引（PAT）标志，4KB 页表项的第 7 位和 4MB 页目录表项的第 12 位

（在 Pentium III处理器中引入的）这个标志用来选择 PAT 项。对于支持页属性表（PAT）的处理器来说，这个标志与 PCD 和 PWT 标志一起，被用来选取 PAT 项，PAT 反过来选择该页的内存类型（见 10.12.“页属性表（PAT）”）。对于不支持 PAT 的处理器，这个位被保留，应该被置为 0。

全局（G）标志，第 8 位

（Pentium Pro 处理器引入的）指明全局页。当一个页被标明为全局的，并且 CR4 中的启用全局页（PGE）标志被置位时，一旦 CR3 寄存器被载入或者发生任务切换，TLB 中的页表或者指向页的页目

录表项并不失效。这个标志可以防止使 TLB 中频繁使用的页（比如操作系统内核或者其它的系统代码）失效。只有软件可以置位或者清零该位。对于指向页表的页目录表项来说，这个标志被忽略。一个页的全局特性是在页表项中设置的。有关更多使用这个标志的信息参看 3.7.“转换后备缓冲区（TLB）”（在 Pentium 和更早期的 IA-32 架构处理器中，该位是被保留的）。

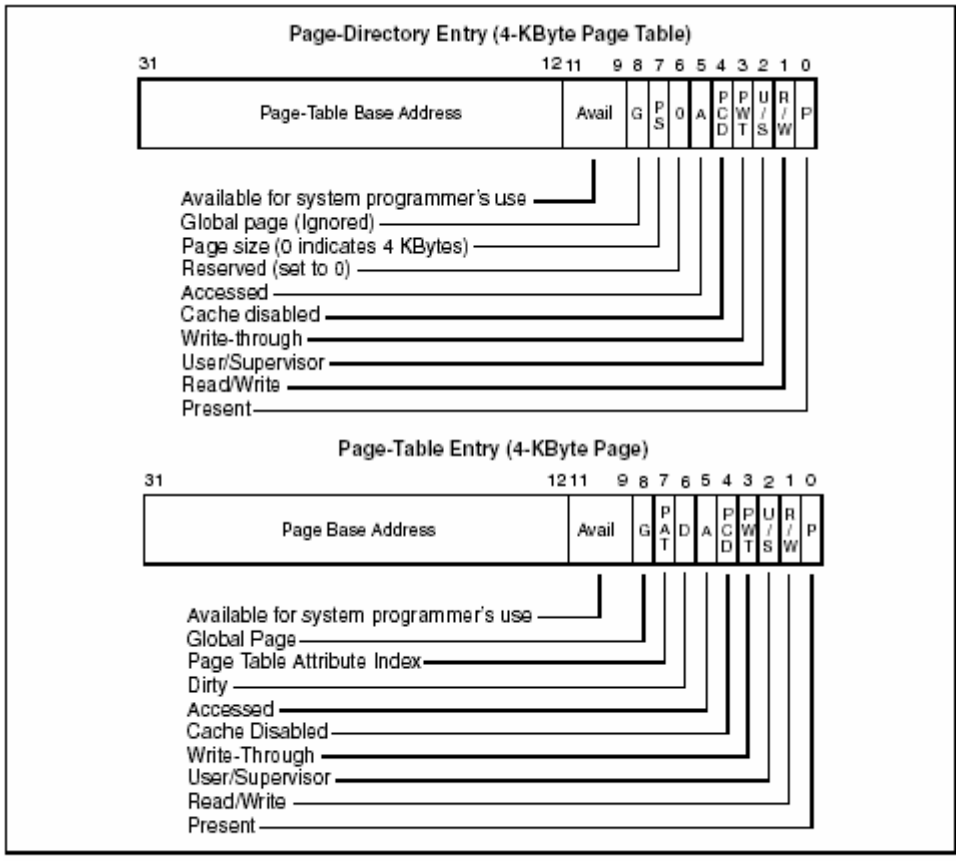

Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

图 3-14 4KB 页和 32 位物理地址页目录表项格式

保留的、可供软件使用的位

对于所有的 IA-32 处理器，第 9、10、11 位都是可以为软件所用。（当“存在位”被清零时，第 1 位到第 31 位都可以为软件使用——见图 3-16。）在指向页表的页目录表项中，第 6 位是被保留的并且应当被置为 0。当控制寄存器 CR4 中的 PSE 和 PAE 标志置位时，如果保留位没有被置为 0，处理器就产生一个页故障。

对于 Pentium Ⅱ及早期的处理器，页表项的位 7 被保留，并置为 0。

对于 4MB 页的页目录表项，第 12 位到第 21 位都是被保留的，并且应当被置为 0。

对于 Pentium III及后来的处理器，4MB 页的页目录表项中，第 13 位到第 21 位都是被保留的，必须被置为 0。

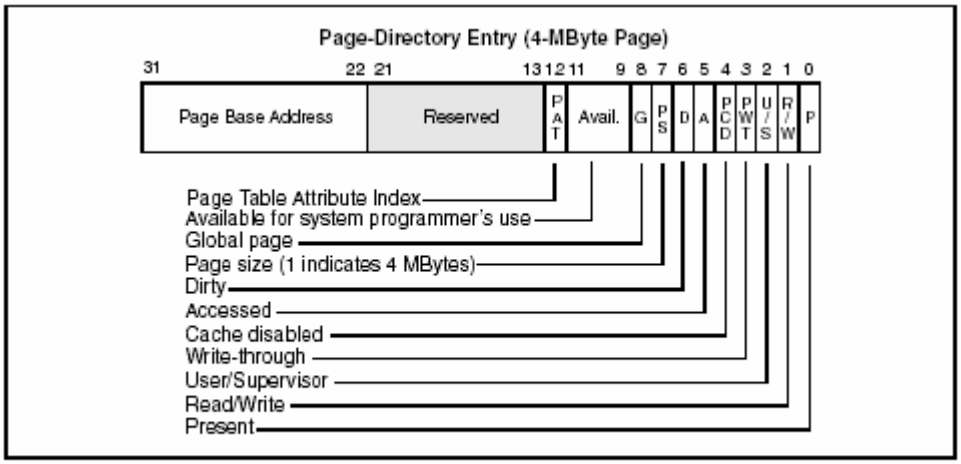

Figure 3-15. Format of Page-Directory Entries for 4-MByte Pages and 32-Bit Addresses

图 3-15 4MB 页和 32 位地址的页目录表项的格式

3.7.7.不存在的页目录表项和页表项

当一个页表项或者页目录表项的“存在”标志被清零时，操作系统可以使用该表项的其余位来存储一些信息，比如该页在磁盘存储系统上的位置（参看图 3-16）。

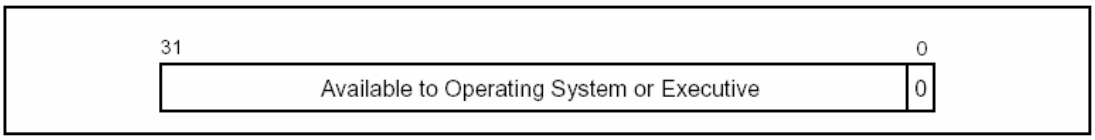

Figure 3-16. Format of a Page-Table or Page-Directory Entry for a Not-Present Page

图 3-16 不存在页的页表项和页目录表项格式

3.8.使用 PAE 分页机制的 36 位物理寻址

PAE 分页机制和 36 位物理寻址能力是从 Pentium Pro 处理器开始引入 IA-32 架构的，通过 CPUID 指令的特性标志 PAE（当 CPUID 指令的源操作数是 2 时，EDX 寄存器的第 6 位就是这个特性标志）来实现的。CR4 中的物理地址扩展（PAE）标志可以开启 PAE 机制，将物理地址从 32 位扩展至 36 位。处理器提供额外的 4 个地址线引脚来容纳这

额外的地址位。使用这个选项必须设置如下标志：

- CR0 寄存器的 PG 标志（第 31 位）——开启分页。
- CR4 寄存器的 PAE 标志（第 5 位）——开启 PAE 分页机制。

当开启 PAE 分页机制时，处理器支持两种尺寸的页：4KB 和 2MB。当使用 32 位寻址时，这两种尺寸的页都能够使用同一个页表集来寻址（也就是说，一个页目录表项可以指向一个 2MB 页，也可以指向一个页表，页表的表项指向 4KB 页）。要支持 36 位的物理地址，分页的数据结构需要做如下的变化：

页表项将变为 64 位以适应 36 位物理地址。每个 4KB 的页目录表和页表也就可以有最多 512 个表项。

一个叫做页目录指针表的新表将被加入到线性地址变换的层次结构中。这个表有 4 个 64 位的表项，放置在层次结构的页目录表之上。随着物理地址扩展机制的开启，处理器支持 4 个页目录表。

寄存器 CR3（PDPR）中的 20 位页目录基地址域被 27 位页目录指针表基地址所替代（见图 3-17）。（此时，寄存器 CR3 叫做 PDPTR。）这个域给出了页目录指针表基地址的高 27 位，迫使页目录指针表的地址是 32 字节对齐的。

于是，线性地址变换得以改变，允许将 32 位的线性地址映射到更大的物理地址空间中。

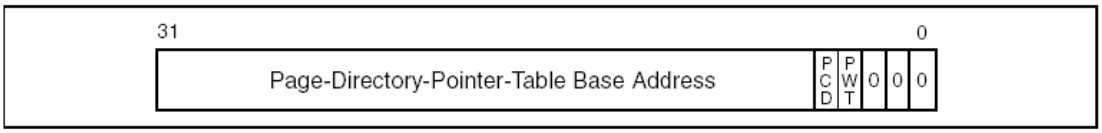

Figure 3-17. Register CR3 Format When the Physical Address Extension is Enabled

图 3-17 开启物理地址扩展时的 CR3 寄存器格式

3.8.1.开启 PAE 的线性地址变换（4KB 页）

当启用 PAE 分页机制时，线性地址是映射到 4KB 页，此时页目录指针表、页目录表和页表的层次结构如图 3-18 所示。这种分页方法可以寻址高达 2^{20} 个页，线性地址空间达 2^{32} 字节（4GB）。

为了选择各种表项，线性地址被分为 3 部分：

- 页目录指针表项——第 30 位和第 31 位，给出页目录指针表项在页目录指针表中的偏移。被选中的表项给出一张页目录表的物理基地址。

- 页目录表项——第 21 位到第 29 位，给出在被选中的页目录表中的偏移。被选择的目录项给出一张页表的物理基地址。
- 页表项——第 12 位到第 20 位，给出在被选中的页表中的偏移。被选中的页表项给出一个页在内存中的物理基地址。
- 页偏移——第 0 位到第 11 位，给出在被选中的页中的偏移。

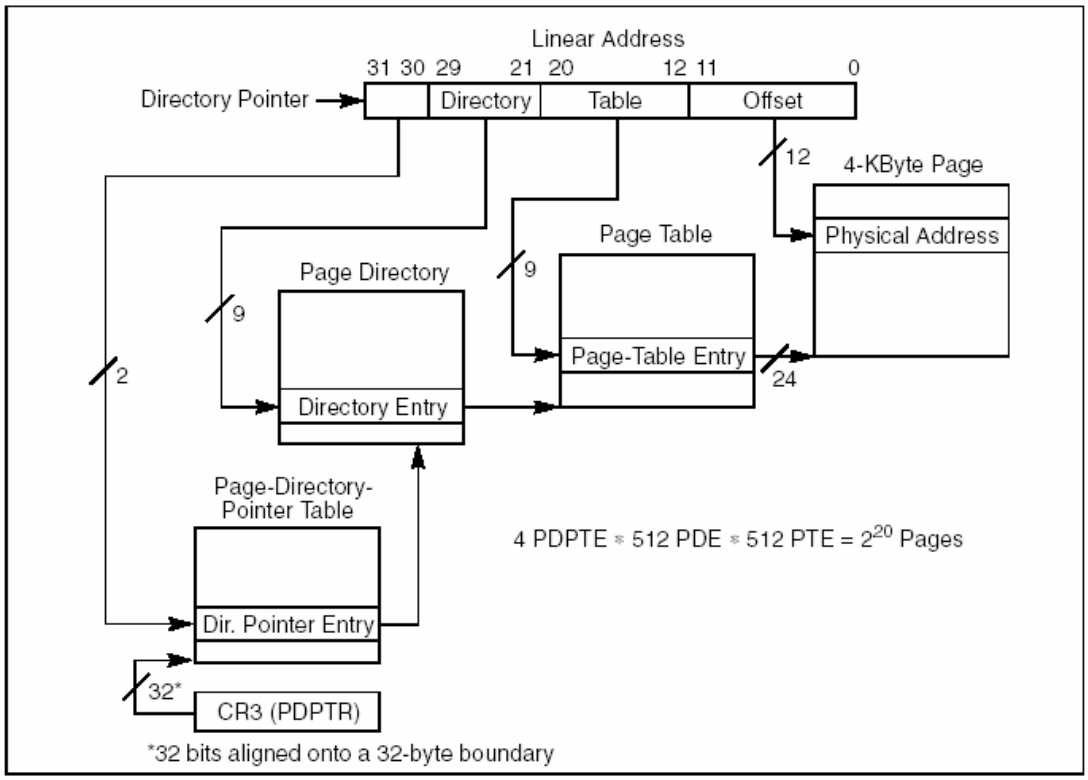

Figure 3-18. Linear Address Translation With PAE Enabled (4-KByte Pages)

图 3-18 开启 PAE 时的线性地址转换（4KB 页）

3.8.2.开启 PAE 的线性地址变换（2MB 页）

图 3-19 显示了当启用 PAE 分页机制时，如何使用页目录指针表和页目录表将线性地址映射到 2MB 页。这种分页方法可以将 2048 个页（4 个页目录指针表项乘上 512 个页目录表项）映射到 4GB 的线性地址空间上。

当启用 PAE 时，通过设置页目录表项中的页尺寸（PS）标志来选择 2MB 页（见图 3-14)。(如表 3-3 中所示，当启用 PAE 时，CR4 寄存器中的 PSE 标志将对页的尺寸不起作用。）一旦 PS 标志被置位，线性地址被分为 3 部分：

- 页目录指针表项——第 30 位和第 31 位给出表项在页目录指针表中的偏移。该页目录指针表项给出一张页目录表的物理基地址。

- 页目录表项——第 21 位到第 29 位提供目录项在页目录表中的偏移。该页目录表项给出了一个 2MB 页的物理基地址。
- 页偏移——第 0 位到第 20 位提供物理地址在页中的偏移。

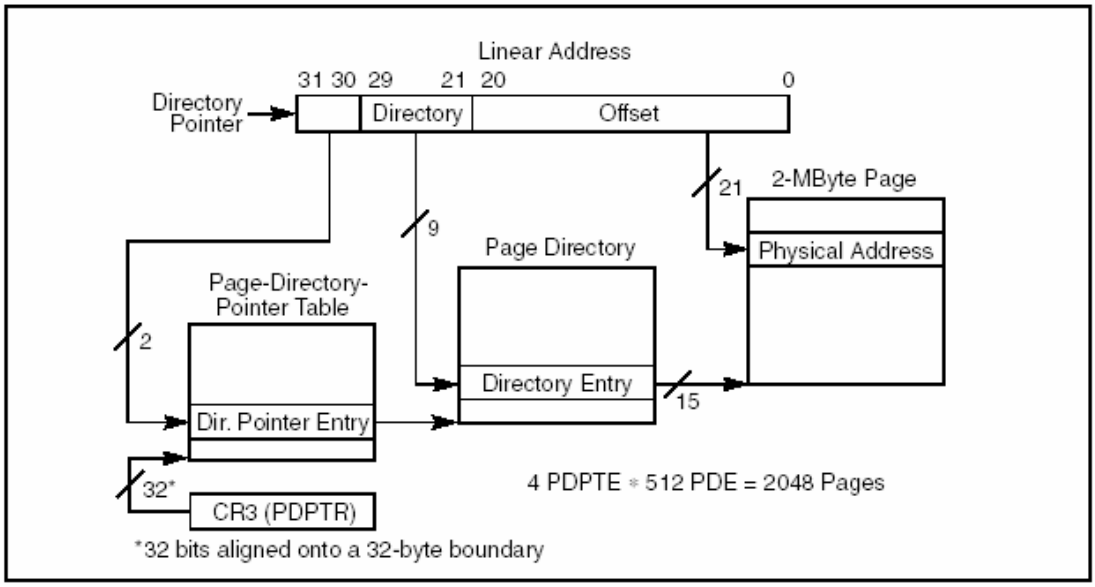

Figure 3-19. Linear Address Translation With PAE Enabled (2-MByte Pages)

图 3-19 PAE 开启时的线性地址转换（2MB 页）

3.8.3. 使用扩展页表结构访问全部扩展物理地址空间

前两节描述的页表结构允许一次性访问 64G 字节扩展物理地址空间中的 4G 字节，物理内存中的另外 4G 字节可以通过下面两种方法之一进行访问：

- 将寄存器 CR3 中的指针改为指向另外一个页目录指针表，这个指针表又指向另外的页目录集和页表集。
- 改变页目录指针表的表项，使其指向其它的页目录表，而这些页目录表指向其它的页表集。

3.8.4. 启用扩展寻址的页目录表项和页表项

使用 4KB 页和 36 位扩展物理地址时，页目录指针表项、页目录表项和页表项的格式如图 3-20 所示。使用 2MB 页和 36 位扩展物理地址时，页目录指针表项和页目录表项的格式如图 3-21 所示。这些表项中标志的功能与 3.7.6.“页目录表项和页表项”中的描述是一样的，其中主要的不同之处如下：

- 增加了页目录指针表项。
- 表项的大小从 32 位增加到了 64 位。
- 页目录表和页表的项数最多为 512 个。
- 表项中的物理基地址域扩展到了 24 位。

注意

当前的IA-32处理器实施PAE机制时，并未在装载页目录指针表项时使用高速缓存。这是与模型相关的行为，而非架构特定行为。未来的 IA-32 处理器会高速缓存页目录指针表项的。

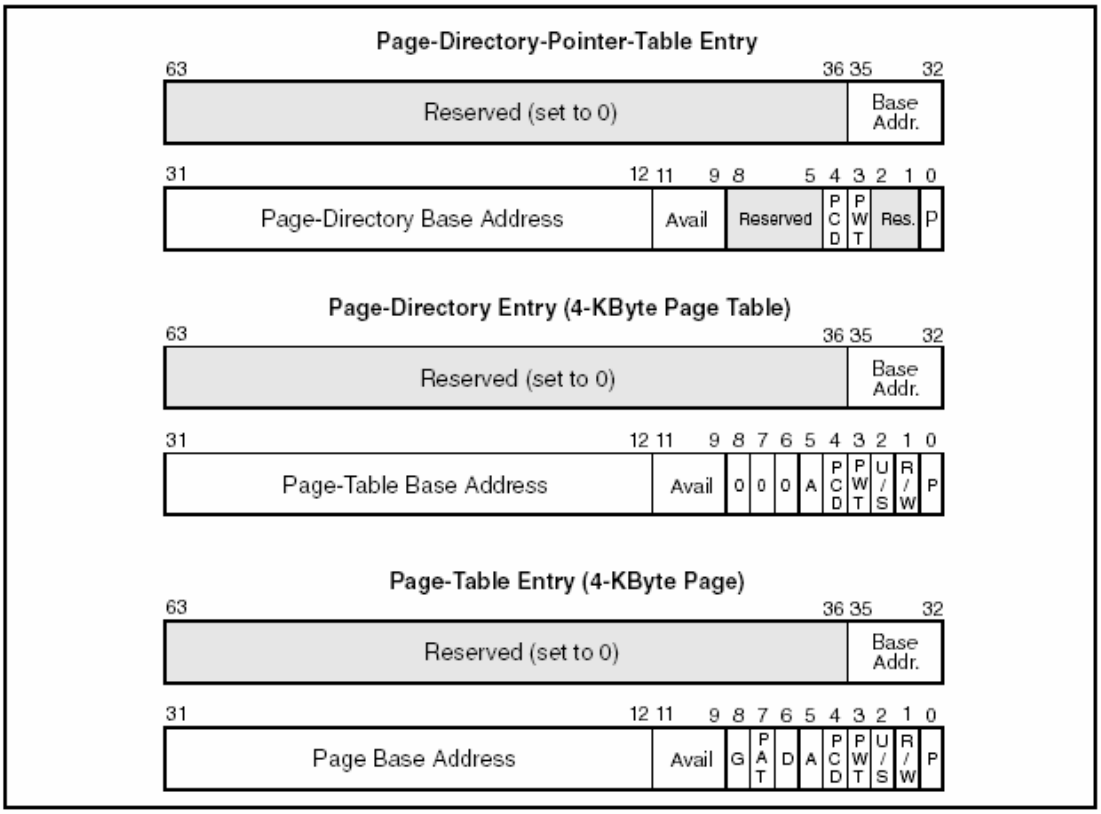

Figure 3-20. Format of Page-Directory-Pointer-Table, Page-Directory, and Page-Table Entries for 4-KByte Pages with PAE Enabled

图 3-20 PAE 开启时的页目录指针表项、页目录表项、页表项的格式（4KB 页）

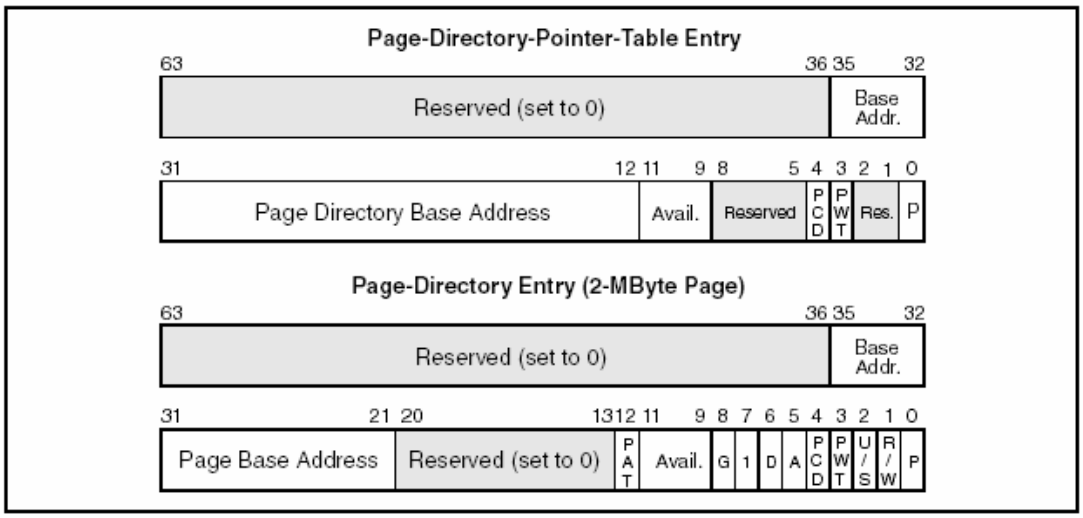

Figure 3-21. Format of Page-Directory-Pointer-Table and Page-Directory Entries for 2-MByte Pages with PAE Enabled

图 3-21 PAE 开启时页目录指针表项、页目录项的格式（2MB 页）

根据表项的不同，表项中的物理基地址的说明如下：

- 页目录指针表项——一张 4KB 页目录表的第一个字节的物理地址。
- 页目录表项——一张 4KB 页表或 2MB 页的第一个字节的物理地址。
- 页表项——一张 4KB 页的第一个字节的物理地址。

在所有的表项中（除了指向 2MB 页的页目录表项之外），基地址都被视为 36 位物理地址的高 24 位，迫使页表和页都是 4KB 对齐的。当页目录表项指向一个 2MB 页时，基地址被视为 36 位物理地址的高 15 位，迫使 2MB 页都是 2MB 对齐的。

页目录指针表项的存在标志位（第 0 位）可以为 0 也可以为 1。如果存在标志被清零，则页目录指针表项余下的位可以为操作系统所用。如果存在标志被置位，那么页目录指针表项就如图 3-20（4KB 页）和图 3-21（2MB 页）所示。

页目录表项中的页尺寸标志（第 7 位）可以判断该表项指向一个页表还是指向一个 2MB 页。当该标志被清零时，则表项指向一个页表；当该标志被置位时，则表项指向一个 2MB 页。此标志使得 4KB 页和 2MB 页可以混在一张页表中。

访问标志（A）（第 5 位）和脏标志（D）（第 6 位）供指向页的表项使用。

所有物理地址扩展表项的第 9、10 和 11 位都可为软件所用。（当“存在位”为 0 时，第 1 到 63 位都可以为软件所用。）图 3-14 中所有被标为“保留”或“0”的位都应当被置为 0 并且不能被软件访问。当控制寄存器 CR4 中的 PSE 和 PAE 标志被置位，而页目录表项和页表项中的保留位没有被置为 0，则处理器产生一个页故障异常（#

PF)；如果页目录指针表项中的保留位没有被置为 0，则处理器会产生一个一般保护异常（＃GP)。

3.9.使用 PSE-36 分页机制的 36 位物理寻址

PSE-36 分页机制提供另一种扩展物理内存寻址到 36 位的方法(与 PAE 机制不同)。这种机制使用页尺寸扩展模式，修改页目录表来映射 4MB 页到 64GB 地址空间。与 PAE 机制一样，处理器提供额外的 4 个地址线引脚来适应增加的地址位。

PSE-36 机制从 Pentium Ⅲ处理器开始引入 IA-32 架构的。这种机制是否可用可以通过 PSE-36 特征位来判断（执行源操作数为 1 的 CPUID 指令后，EDX 寄存器的第 17 位)。

如表 3-3 所示，如下的标志必须被设置或者清零来启用 PSE-36 分页机制：

- PSE-36 CPUID 特征标志——置位时，则表示执行 CPUID 指令时显示可以使用 PSE-36 分页机制。
- CR0 寄存器中的 PG 标志（第 31 位）——置为 1 来启用分页。
- CR4 寄存器中的 PAE 标志（第 5 位）——置为 0 来禁用 PAE 分页机制。
- CR4 寄存器中的 PSE 标志（第 4 位）和 PDE（页目录表项）中的 PS 标志——置为 1 来启用使用 4MB 页的页尺寸扩展。
- 或者 CR4 寄存器中的 PSE 标志（第 4 位）——置为 1，并且 PDE 中的 PS 标志——置为 0，来使用 32 位寻址的 4KB 页（4GB 以下)。

图 3-22 显示了如何用扩展的页目录表项将 32 位线性地址映射到 36 位物理地址。这里，线性地址分为两部分：

- 页目录表项——第 22 位到第 35 位，给出表项在页目录表中的偏移。该表项给出一个 36 位地址的高 14 位以确定一个 4MB 页的物理基地址。
- 页偏移——第 0 位到第 21 位给出一个物理地址在页中的偏移。

这种分页方法可以将 1024 个页映射到 64GB 物理地址空间。

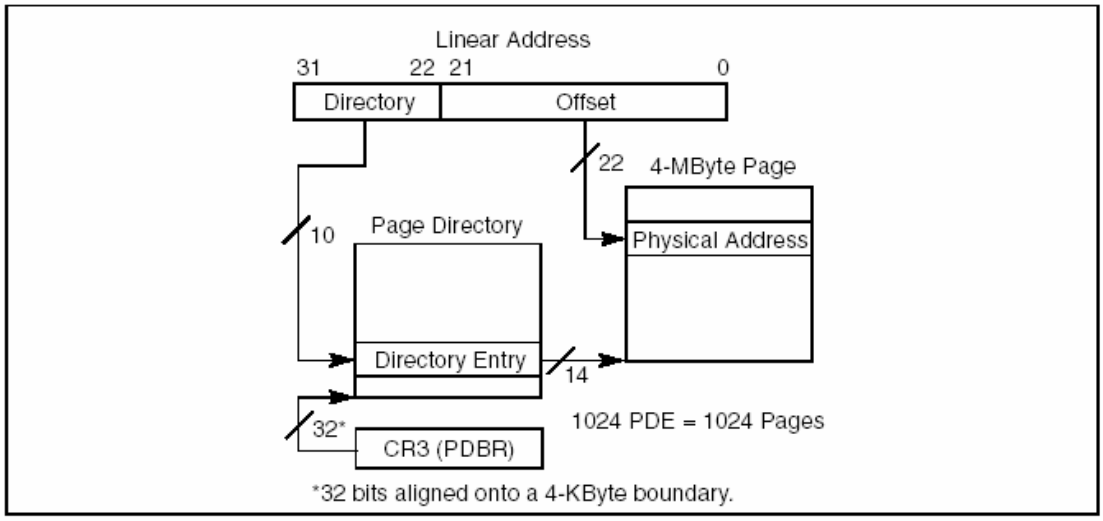

Figure 3-22. Linear Address Translation (4-MByte Pages)

图 3-22 线性地址转换（4MB 页）

图 3-23 显示了使用 4MB 页和 36 位物理地址时的页目录表项的结构。3.7.6.“页目录表项和页表项”描述了第 0 位到第 11 位各个标志和域的功能。

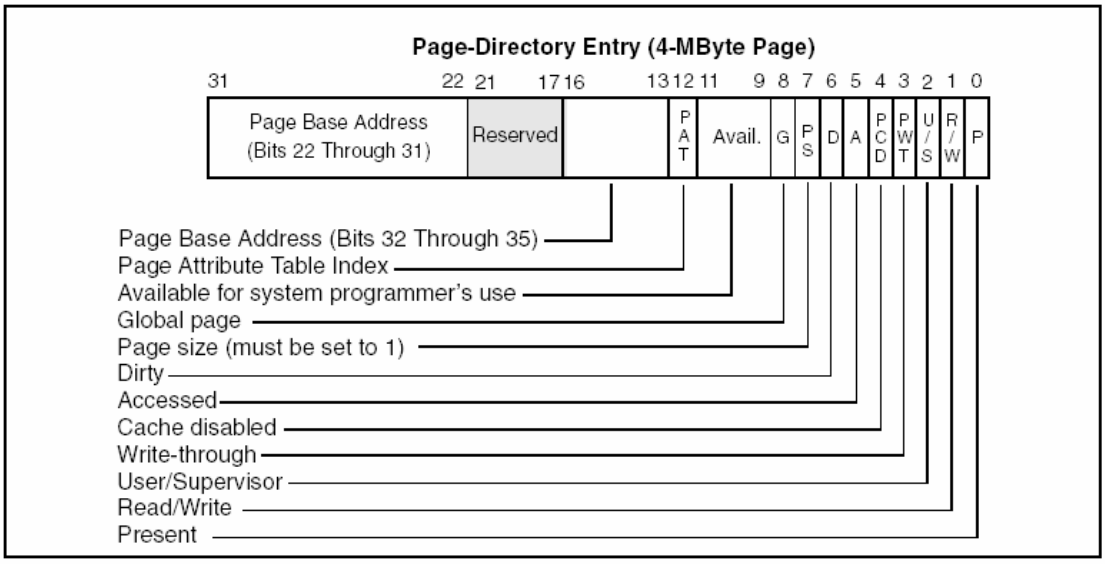

Figure 3-23. Format of Page-Directory Entries for 4-MByte Pages and 36-Bit Physical Addresses

图 3-23 4MB 页和 36 位物理地址的页目录表项的格式

3.10.段到页的映射

IA-32 架构的分段和分页机制提供了很多方法来进行内存管理。当分段和分页联合使用时，有好几种方法可以将段映射到页。比如说，在平坦寻址环境（不分段）中，代码、数据和堆栈模块可被映射到一个或者多个段（最大为 4GB）中，这些段共享同一

个线性地址范围（见图 3-2）。本质上来说，段对应用程序和操作系统或者管理程序是不可见的。如果使用分页，分页机制可以将单一的线性地址空间（只有一个段）映射到虚拟内存。每个进程（或任务）都可以有自己的大线性地址空间（包含在它自己的段中），通过它自己的页目录表和一组页表将线性地址空间映射到虚拟内存。

段可以小于页的大小。如果某个此种类型的小段被放置在一个不与其它段共享的页内，该页剩余的内存就浪费了。比如，一个一字节信号量这样的小数据结构，如果它自己独占一个页，就将占据 4K 字节的空间。如果使用多个信号量，将它们打包以后放在一个页内会更加高效。

IA-32 架构并不强制规定页边界与段边界之间的对应关系。一个页可以包含一个段的结尾和另外一个段的开始。与之对应的是，一个段可以包含一个页的结尾和另外一个页的开始。

强迫页边界和段边界的对齐，内存管理软件就会得以简化和更加高效。比如，如果可以放置在一个页中的段被放置在两个页中，那么，访问这个段所需的分页工作就会成倍提高。

如图 3-24 所示，让每个段都有自己的页表是一种结合分页和分段机制，简化内存管理软件的方法。这个方法让每个段在页目录表中有一个单独的表项，提供将整个段分页的访问控制信息。

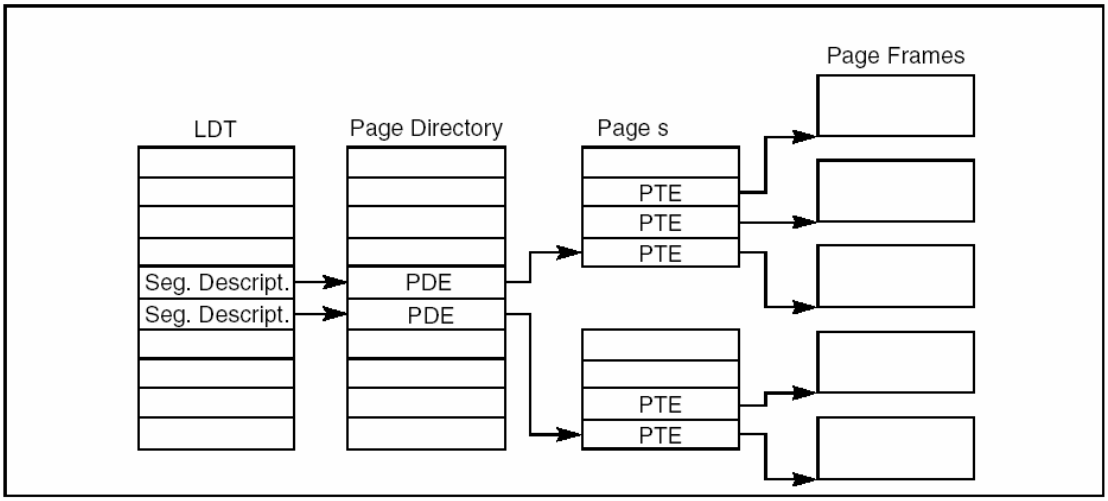

Figure 3-24. Memory Management Convention That Assigns a Page Table to Each Segment

图 3-24 把一个页表指派给一个段的内存管理惯例

3.11.转换后备缓冲区（TLB）

处理器将最近最常使用的页目录表项和页表项存放在 on－chip 高速缓存中，这种高速缓存叫做转换后备缓冲区（缩写为“TLB”）。P6 系列和 Pentium 处理器中，数据和指令有各自的 TLB。而且，P6 系列处理器中，4KB 和 4MB 的页有各自的 TLB。CPUID 指令可以用来得知 P6 系列和 Pentium 处理器的 TLB 的大小。

大多数分页操作都是使用 TLB 中的内容来进行的。仅当进行地址转换所需的请求页不在 TLB 中时，才需要花费总线周期去访问内存中的页目录表和页表。

应用程序和一般任务（特权级大于 0）是无法访问 TLB 的。也就是说，应用程序不能使 TLB 失效。只有操作系统和特权级为 0 的管理程序可以使 TLB 失效或者让选定的 TLB 项失效。每当一个页目录表项或者页表项有变化时（包括“存在”标志被设置为 0），操作系统都要立即将 TLB 中的相应项变为无效，这样，当下次引用这个项的时候，可以更新它。

每当装载 CR3 寄存器时，所有的（非全局的）TLB 都自动失效（除非某个页或者页表项的 G 标志被置位，这个在本节后续部分有描述）。有两种方式来装载 CR3 寄存器：

- 显式地，使用 MOV 指令，比如：

  MOV CR3，EAX

  在这里 EAX 寄存器包含有一个合适的页目录基址。
- 隐式地，通过执行任务切换，它会自动改变 CR3 寄存器的内容。

INVLPG 指令是用来使 TLB 中某个页表项失效的。通常，这个指令只是使单个 TLB 项失效。然而，在某些情况下，这条指令不仅仅让所指定的项失效，甚至能让所有的 TLB 项失效。这条指令会忽略页目录表项或者页表项的 G 标志（参看下一段落）。

（在 Pentium Pro 处理器引入的。）为了避免在任务切换或者装载 CR3 寄存器时，TLB 中频繁使用的页被自动置为失效，可以使用 CR4 寄存器中的“启用全局页（PGE）”标志和页目录表项或页表项中的“全局（G）”标志（第 8 位）。（更多关于“全局”标志的信息参看 3.7.6.“页目录表项和页表项”。）当处理器为一个全局页装载一个页目录表项或者页表项到 TLB 时，这个表项将以不确定的方式保留在 TLB 中。能够肯定地使全局页表项失效的方法如下：

- 清除 PGE 标志，然后使 TLB 失效。
- 执行 INVLPG 指令来使 TLB 中的单个页目录表项或页表项失效。

更多关于使 TLB 失效的信息参看 10.9.“使 TLB 失效”。

第 4 章 保　　护

在保护模式下，IA-32 架构提供的保护机制包括段和页两级操作层次。根据特权级别（段的特权级别有 4 级，页有 2 级）的不同，保护机制提供了对特定段或者页进行限制性访问的能力。比如，可以将重要的操作系统代码和数据放在特权级更高的段（相比于包含应用程序的段来说）上来保护它们，这样，处理器的保护机制就可以防止应用程序代码不加控制地访问操作系统的代码和数据。

在软件开发的任何阶段，都可以使用段和页的保护机制来定位和检测设计中的问题和错误。这种机制可以整合到最终产品中，以加强操作系统、工具软件和应用软件的健壮性。

当使用保护机制时，对内存的任何引用都要进行检验，以确定是否符合各种保护性要求。这些检验都是在内存周期开始之前进行的，任何违例都会导致异常的产生。这种检验都是与地址转换同时进行的，不会对性能造成什么影响。保护性检验可以分为如下几类：

- 界限检验。
- 类型检验。
- 特权级检验。
- 可寻址区域的限制。
- 例程入口点的限制。
- 指令集的限制。

所有的保护违例都会产生异常。关于异常机制的解释参看第五章“中断和异常处理”。本章描述保护机制和导致异常的违例。

以下几节描述保护模式下提供的保护机制。有关实地址模式和虚拟 8086 模式保护的更多信息，参看第十六章“8086 仿真”。

4.1.启用/禁用段保护和页保护

把 CR0 寄存器中的 PE 标志位置位，就使处理器切换到保护模式，从而启用了段保护机制。一旦进入保护模式就没有办法重新设置来关闭保护机制或者打开保护机制。

在保护模式下，可以将基于特权级的段保护机制事实上关闭，即将所有的段选择子和段描述符的特权级都设为 0（最高特权级）。这个办法可以消除段之间的特权级保护屏障，但是其它的保护检验如界限检验和类型检验仍然要进行。

启用分页（置位 CR0 寄存器中的 PG 标志位）以后，页级保护也就自动开启了。同样，一旦分页启用了，就没有控制位可以将页级保护关闭。然而，通过如下操作可以禁用页级保护：

- 清除 CR0 寄存器中的 WP 标志。
- 将每个页目录表项和页表项的读/写（R/W）标志位和用户/管理（U/S）标志位都置位。

这种做法将每个页都变成了可写的、用户态的，实际上关闭了页级保护。

4.2.用于段级和页级保护的域和标志

处理器的保护机制使用如下系统数据结构中的域和标志来控制对段和页的访问：

- 描述符类型（S）标志——（段描述符的第二个双字的第 12 位）确定段描述符的类型是系统段描述符、代码段描述符还是数据段描述符。
- 类型域——（段描述符的第二个双字的第 8 到第 11 位）确定段的类型是代码段、数据段还是系统段。
- 界限域——（段描述符的第一个双字的第 0 到第 15 位，第二个双字的第 16 到第 19 位）这个域和 G 标志、E 标志（对数据段而言）一起确定段的长度。
- G 标志——（段描述符的第二个双字的第 23 位）与界限和 E 标志（对数据段而言）一起决定段的长度。
- E 标志——（数据段描述符的第二个双字的第 10 位）与界限域和 G 标志一起决定段的长度。
- 描述符特权级（DPL）域——（段描述符的第二个双字的第 13 和第 14 位）确定段的特权级。
- 请求特权级（RPL）域——（段选择子的第 0 和第 1 位）确定段选择子的请求特权级。
- 当前特权级（CPL）域——（CS 段寄存器的第 0 和第 1 位）指明当前执行程序/例程的特权级。术语“当前特权级（CPL）”就是指这个域的设置。

- 用户/管理（U/S）标志——（页目录表项或页表项的第 2 位）确定页的类型：用户态还是管理态。
- 读/写（R/W）标志——（页目录表项和页表项的第 1 位）确定页的访问类型：只读还是可读可写。

图 4-1 显示了各个域和标志在数据、代码和系统等段描述符中的位置；图 3-6 显示了 RPL（或 CPL）域在段选择子（或 CS 寄存器）中的位置；图 3-14 显示了 U/S 和 R/W 标志在页目录表项和页表项中的位置。

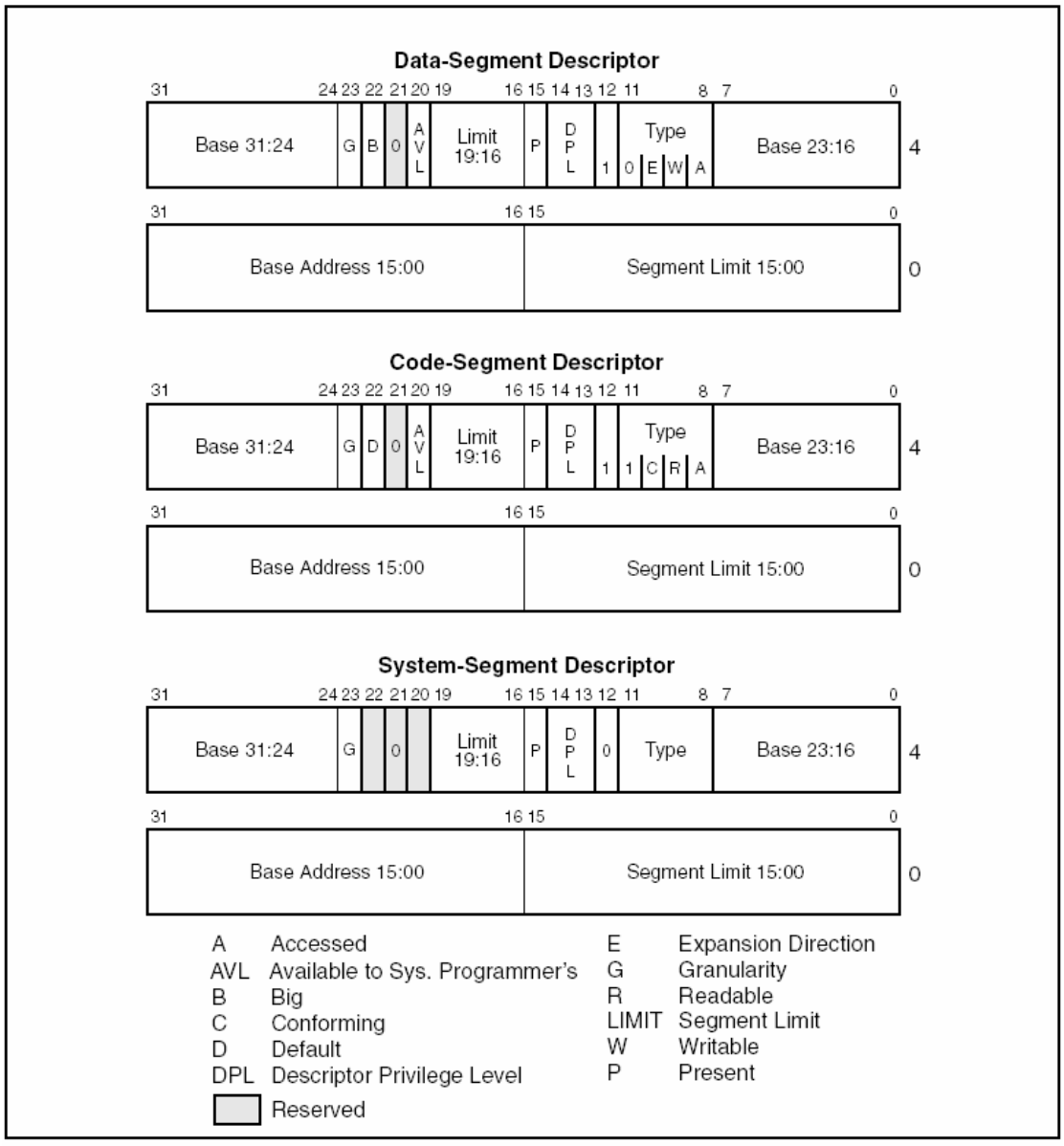

Figure 4-1. Descriptor Fields Used for Protection

用这些域和标志可以实现多种不同风格的保护方案。当操作系统或者管理程序创建一个描述符时，会在这些域和标志中放置相应的值，以维持与所选择的特定保护风格的一致性。应用程序一般不会访问和改变这些域和标志。

下面各节将描述处理器如何使用这些域和标志来完成在本章前面介绍中描述的各种检验。

4.3. 界限检验

段描述符中的界限域用来防止程序/例程在段之外的内存寻址。界限的有效值取决于粒度 G 标志的设置（见图 4-1）。对于数据段，界限还要依赖于扩展方向 E 标志和 B 标志（的设置）。当段描述符的类型是数据段时，E 标志是类型域中的一个位。

当 G 标志为 0（粒度为字节）时，段的有效界限就是段描述符中 20 位界限域中的值。此时，界限的范围为 0 到 FFFFFH（1MB）。当 G 标志为 1 时（4KB 页粒度），处理器将界限域的值乘上一个因子 2^{12}（4KB）作为段界限的有效值。这种情况下，有效界限的范围为 FFFH（4KB）到 FFFFFFFFH（4GB）。注意，当使用乘数因子（G 标志为 1）时，就不需要对段偏移的低 12 位进行界限检验。比如，如果段界限为 0 时，偏移从 0 到 FFFH 仍然是有效的。

除了向下扩展的数据段之外，所有段的有效界限就是段中被允许访问的最后地址，它比段的长度小 1 个字节。任何时候，试图访问段中如下地址时，处理器都将产生一个一般保护异常：

- 大于有效界限的字节偏移。
- 大于有效界限减 1 的字偏移。
- 大于有效界限减 3 的双字偏移。
- 大于有效界限减 7 的四字偏移。

对于向下扩展的数据段而言，段界限有同样的功能，但是意义却不一样。此处，有效界限定义为段中最后一个不允许访问的地址。如果 B 标志为 1，合法偏移的范围为有效界限加 1 到 FFFFFFFFH；如果 B 标志为 0，合法偏移的范围为有效界限加 1 到 FFFFH。当段界限为 0 时向下扩展段的长度最大。

界限检验可以捕捉到某些程序的错误，比如失控的代码和脚本，以及非法的指针计算。这些错误一旦发生就可以被检测到，因此原因的确认就更加容易。没有界限检验，这些错误可能会导致另外一个段的代码和数据被覆盖。

除了检验段界限外，处理器还要检验描述符表的界限。GDTR 和 IDTR 寄存器含有 16 位的界限值，处理器使用这个值防止进程从各自描述符表之外的地方读取段描述符。

LDTR 和任务寄存器含有 32 位的界限值(分别从当前的 LDT 段描述符和 TSS 中读取的)。处理器使用这些段界限来阻止对超越当前 LDT 和 TSS 界限的访问。更多关于 GDT 和 LDT 界限域的信息参看 3.5.1.“段描述符表”；更多关于 LDT 界限域的信息参看 5.10.“中断描述符表”；更多关于 TSS 段界限域的信息参看 6.2.3.“任务寄存器”。

4.4 类型检验

段描述符中有两个地方含有类型信息：

- S（描述符类型）标志。
- 类型域。

处理器使用这些信息来检测某些编程方面的错误，这些错误可能导致不正确的或者无意识的使用一个段或者门。

S 标志指明描述符的类型是系统、代码还是数据。类型域提供了额外的 4 个位来定义各种类型的代码、数据和系统等描述符。表 3-1 显示了对代码描述符和数据描述符类型域的各种组合的解释；表 3-2 显示了对系统描述符类型域的各种组合的说明。

每当对段选择子和段描述符进行操作时，处理器就检验类型信息。下面列举了进行类型检验的几个典型操作的例子，这几个例子没有穷尽所有情况。

- **当把段选择子装入段寄存器时**。特定的段寄存器只能容纳特定类型的描述符。比如：

  ——CS 寄存器只能装入代码段的选择子。

  ——不可读的代码段或者系统段的选择子不能载入数据段寄存器（DS、ES、FS 和 GS)。

  ——只有可写的数据段的选择子才能装入 SS 寄存器。

- **当段选择子装入 LDTR 或任务寄存器时。**

  ——LDTR 只能装入 LDT 的选择子。

  ——任务寄存器只能装入 TSS 的段选择子。

- **当指令访问其描述符已经装入段寄存器的段时**。特定的段只能以某些预定的方式访问，比如：

  ——禁止指令写入一个可执行的段。

  ——禁止指令写入一个不可写的数据段。

——除非可读标志置位，否则禁止指令读一个可执行段。

- **当指令的操作数含有段选择子时。**某些指令只能访问某个特定类型的段或者门，比如：

  ——远调用（CALL）或远跳转（JMP）指令只能访问一致性代码段、非一致性代码段、调用门、任务门或者 TSS 的段描述符。

  ——LLDT 指令只能引用指向 LDT 的段描述符。

  ——LTR 指令只能引用指向 TSS 的段描述符。

  ——LAR 指令只能引用指向 LDT、TSS、调用门、任务门、代码段或数据段的描述符。

  ——LSL 指令只能引用 LDT、TSS、代码段或者数据段的描述符。

  ——IDT 表项只能是中断门、陷阱门或任务门。

- **在某些内部操作期间。比如：**

  ——在远调用或远跳转（执行远 CALL 或远 JMP 指令）期间，通过检验 CALL 或 JMP 指令操作数中的段（或者门）选择子指向的段（或者门）描述符的类型域，处理器来确定即将执行的控制转移（通过门调用或者跳转，或者切换任务，调用或者跳转到另外一个代码段）。如果描述符类型是代码段或调用门，表明是调用或者跳转到另外一个代码段；如果描述符类型是 TSS 或者任务门，表明是进行任务切换。

  ——通过调用门进行的调用或者跳转（或者通过陷阱或中断门调用中断或异常处理程序）时，处理器自动对门所指向的段描述符进行检验，以确定描述符类型是否是一个代码段。

  ——通过任务门调用或跳转至一个新任务（或者通过任务门调用中断或异常处理程序至一个新任务）时，处理器自动检验任务门所指向的段描述符是否为一个 TSS。

  ——通过直接访问指向 TSS 的索引进行调用或者跳转到一个新任务时，处理器自动检验 CALL 或 JMP 指令所指向的段描述符是否是一个 TSS。

  ——从一个嵌套的任务返回时（通过 IRET 指令），处理器会检验当前 TSS 中的前一个任务链接域是否指向一个 TSS。

4.4.1.空段选择子的检验

试图将一个空段选择子（见 3.4.1.“段选择子”）装入 CS 段寄存器或 SS 段寄存器会产生一个一般保护异常（＃GP）。可以将空段选择子装入 DS、ES、FS 或 GS 等寄存器，但是任何试图通过值为空段选择子的寄存器来访问段的操作，都将产生一般保护异常（＃GP）异常。将空段选择子载入不使用的数据段寄存器，可以很好地检测出对未使用的段寄存器的访问或者防止对数据段的意外访问。

4.5.特权级

处理器的段保护机制识别 4 个特权级别，编号为 0 到 3。数值越大，特权越少。图 4-2 显示，处理器如何以保护环的方式来解释这些特权级别的。

环的中心（保留给特权级最高的代码、数据和栈）供包含重要软件的段使用，通常是操作系统内核。外面的环给不是很重要的软件使用。（仅使用 2 个特权级的系统应该使用特权 0 和特权 3。）

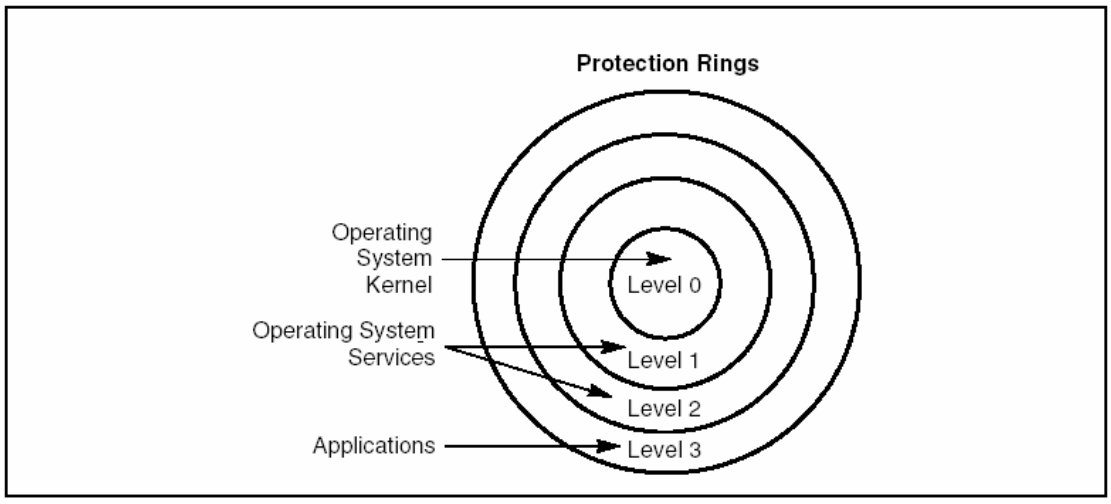

Figure 4-2. Protection Rings

除非某些可控制情况之外，处理器使用特权级来阻止较低特权级的进程或任务访问特权级较高的段。当处理器检测到一个特权违例时，就会产生一个一般保护异常（#GP）。

为了进行代码段和数据段之间的特权级检验，处理器识别以下三种类型的特权级：

- **当前特权级（CPL）。**CPL 是当前执行程序或任务的特权级。它存在 CS 段寄存器和 SS 段寄存器的第 0 位和第 1 位中。一般地，CPL 与当前指令所在代码段

的特权级相等。当进程的控制流程转到一个不同特权级的代码段时，处理器就改变 CPL。当访问一致性代码段时，对 CPL 的处理会略有不同。一致性代码段可以被任何数值上等于或者大于(特权较低)本一致性代码段 DPL 的代码访问。同样，当处理器访问一个与 CPL 特权级不一样的一致性代码段时，不改变 CPL。

- **描述符特权级（DPL）。**DPL 是段或门的特权级。它存储在段或门的描述符的 DPL 域中。一旦当前执行的代码段试图访问一个段或门时，处理器会将那个段或门的 DPL 与 CPL 以及那个段或门的选择子的 RPL（本节后面进行描述）进行比较。根据所访问的段或门的类型，对 DPL 的解释也会不一样：

  ——**数据段。**DPL 指明访问这个段的进程或任务的特权级可以具有的最大数值。比如，如果某个数据段的 DPL 是 1，只有运行在 CPL 为 0 或 1 的进程才可以访问它。

  ——**非一致性代码段（不使用调用门）。**DPL 指明访问这个段的进程或任务应该具有的特权级。比如，某个非一致性代码段的 DPL 为 0，那么只有 CPL 为 0 的进程才能访问它。

  ——**调用门。**DPL 指明能够访问这个调用门的当前进程或者任务的特权级应该具有的最大数值。（这个访问规则同样适用数据段。）

  ——**通过调用门访问的一致性代码段和非一致性代码段。**DPL 指明能访问这个段的进程或任务的特权级的最低数值。比如，一个一致性代码段的 DPL 为 2，那么 CPL 为 0 或 1 的进程就不能访问它。

  ——**TSS。**DPL 指明能够访问这个 TSS 的当前进程或者任务应该具有的特权级的最高数值。（这个访问规则同样适用数据段。）

- **请求特权级（RPL）。**RPL 是赋给段选择子的取代性特权级，存储在段选择子的第 0 位和第 1 位。处理器检验 RPL 的同时也检验 CPL 来决定对段的访问是否被允许。即使请求访问的进程或任务有足够的特权（也就是说 CPL 权限够了——译注）去访问一个段，但是，如果 RPL（即指向将要去访问的这个段的段描述符的段选择子中的 RPL——译注）的特权级不够，访问会被拒绝。也就是说，如果这个段选择子的 RPL 在数值上大于 CPL，RPL 就取代 CPL，反之亦然。RPL 可用来确保特权代码不代表应用程序去访问的一个段，除非这个应用程序本身有这个段的访问特权。更详细的关于 RPL 的用途和典型用法参看 4.10.4.“检验调用者访问特权（ARPL 指令)”。

特权级检验是在段描述符的段选择子被装入段寄存器时进行的。用于数据访问的检验不同于用于代码段的进程控制转移的检验，因而，后续几节将分别讨论这两种访问。

4.6.访问数据段时的特权级检验

为了访问数据段中的操作数，就必须将该数据段的段选择子装入数据段寄存器（DS、ES、FS 或 GS）或者堆栈段寄存器（SS）。(可以用 MOV、POP、LDS、LES、LFS、LGS 和 LSS 等指令装载段寄存器。)处理器将段选择子装入段寄存器之前，通过比较当前运行程序或任务的特权级（CPL)、段选择子的 RPL 和段描述符的 DPL 来进行特权级检验（见图 4-3)。如果 DPL 在数值上大于或者等于 CPL 和 RPL，则处理器将段选择子装入段寄存器。否则，处理器会产生一个一般保护异常，不装载段寄存器。

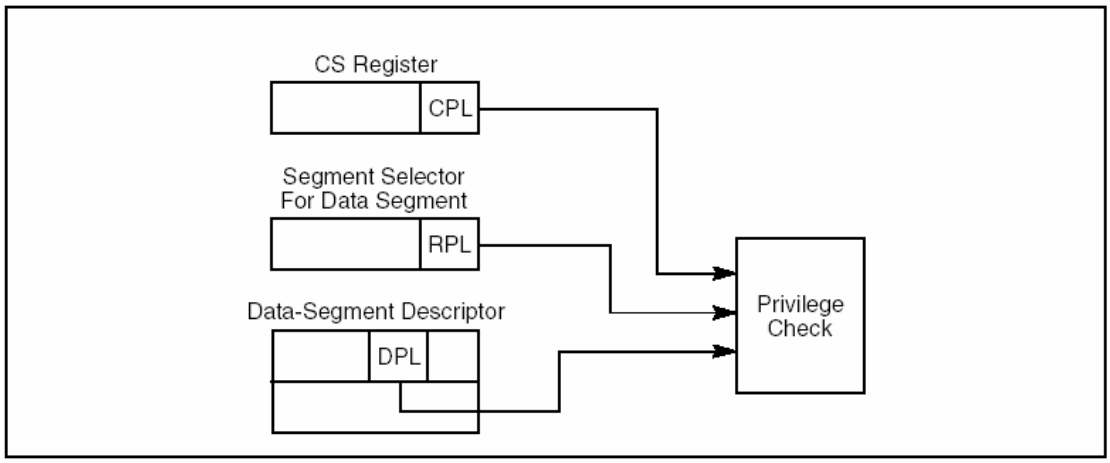

Figure 4-3. Privilege Check for Data Access

如图 4-4 所示，4 个进程（在代码段 A、B、C、D 中）分别运行在不同的特权级，都试图访问同一个数据段。

- 代码段 A 的进程可以通过段选择子 E1 来访问数据段 E，因为代码段 A 的 CPL 和段选择子 E1 的 RPL 与数据段 E 的 DPL 相等。
- 代码段 B 的进程可以通过段选择子 E2 来访问数据段 E，因为代码段 B 的 CPL 和段选择子 E2 的 RPL 都在数值上小于数据段 E 的 DPL（也就是 CPL 和 RPL 具有更高的特权级)。代码段 B 的进程也可以通过段选择子 E1 来访问数据段 E。
- 代码段 C 的进程不能通过段选择子 E3（虚线）访问数据段 E，因为代码段 C 的 CPL 和段选择子 E3 的 RPL 在数值上都大于数据段 E 的 DPL（意味着较小的特权级)。即使代码段 C 的例程打算使用段选择子 E1 或 E2，这样 RPL 可被接

受，但是 CPL 的级别不够，仍然不能访问数据段 E。

- 代码段 D 的进程本应当可以访问数据段 E，因为代码段 D 的 CPL 在数值上小于数据段 E 的 DPL。然而段选择子 E3 的 RPL 在数值上大于数据段 E 的 DPL，因此该进程对数据段 E 的访问被禁止了。如果代码段 D 的进程使用段选择子 E1 或 E2 来访问数据段，那么对数据段 E 的访问就是被允许的。

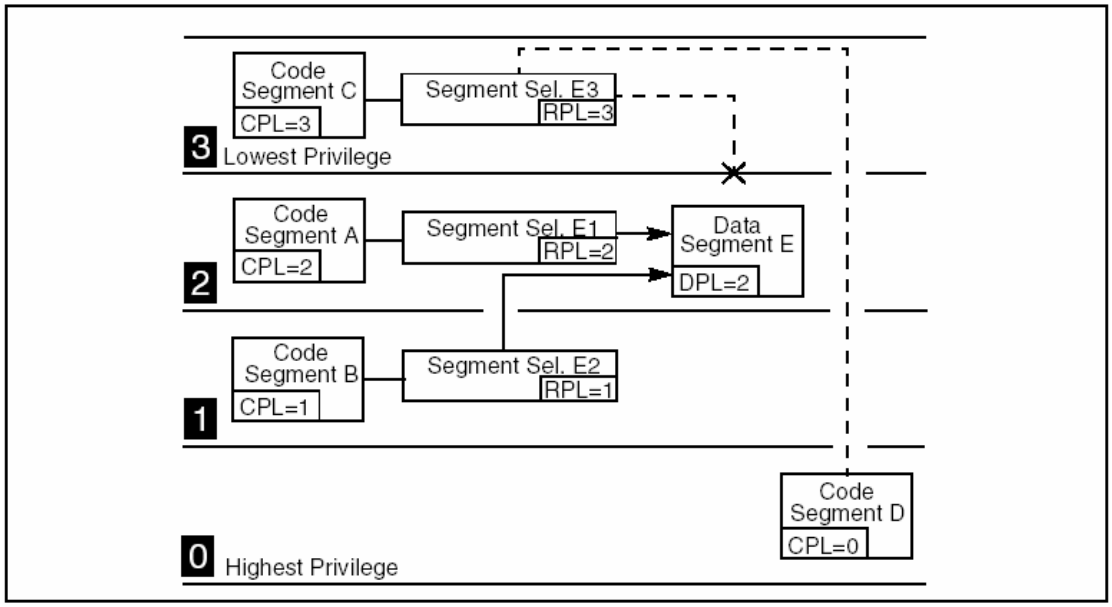

Figure 4-4. Examples of Accessing Data Segments From Various Privilege Levels

如前面的例子所示，一个进程或者任务的可寻址范围随着它的 CPL 变化而变化。当 CPL 为 0 时，任何特权级的数据段都是可访问的；当 CPL 为 1 时，只能访问特权级为 1 到 3 的数据段；当 CPL 为 3 时，只能访问特权级为 3 的数据段。

段选择子的 RPL 通常可以改变进程或者任务的可寻址范围。如果使用得当，RPL 可以防止低特权的进程或任务有意无意地错误使用特权数据段的段选择子而造成的问题。

值得注意的是，数据段段选择子的 RPL 是软件控制的。比如，一个 CPL 为 3 的应用程序可以将数据段的段选择子的 RPL 置为 0。一旦 RPL 设为 0，只有 CPL 检验而不是 RPL 检验，将提供针对数据段特权级的安全保护，以防止故意的直接的违例。为了防止这种类型的特权级检验违例，无论何时，一个进程或任务从另外一个进程接收到一个数据段选择子，都要进行访问特权级的检验（参看 4.10.4.“检验调用者访问特权（ARPL 指令）”）。

4.6.1.访问代码段中的数据

某些情况下，需要访问在代码段中的数据结构。可以用下述方法访问代码段中的数据：

- 将一个非一致性的、可读代码段的段选择子装入数据段寄存器。
- 将一个一致性的、可读代码段的段选择子载入数据段寄存器。
- 使用 CS 前缀来强制访问选择子已经装入 CS 寄存器的可读代码段。

方法 1 的规则同样适用于访问数据段。方法 2 总是正确的，因为无论 DPL 是多少，一致性代码段的特权级实际上与 CPL 是相等的。方法 3 也总是正确的，因为 CS 寄存器选中的代码段的 DPL 与 CPL 是相等的。

4.7.装载 SS 寄存器时的特权级检验

当堆栈段的段选择子被装入 SS 寄存器时，也会进行特权级检验。所有与堆栈段相关的特权级必须与 CPL 匹配，也就是说，CPL、堆栈段的段选择子的 RPL 和段描述符的 DPL 必须相等。如果 RPL 和 DPL 不等于 CPL，就会产生一个一般保护异常（#GP）。

4.8.在代码段之间进行程序控制转移时的特权级检验

程序控制从一个代码段转换到另一个代码段时，必须把目标代码段的段选择子装入 CS 寄存器。在装载过程中，处理器会检查目标代码段的段描述符，对段界限、类型及特权级进行检验。一旦通过检验，就装载 CS 寄存器，转移程序控制到新的代码段，从 EIP 寄存器所指的指令开始程序的执行。

进程控制转移是用 JMP、CALL、RET、SYEENTER、SYSEXIT、INT n 和 IRET 指令或者异常和中断机制来实现的。异常、中断和 IRET 指令作为特例将在第 5 章“中断和异常处理”中讨论。本章仅讨论 JMP、CALL、RET、SYSENTER 和 SYSEXIT 指令。

JMP 或 CALL 指令可以通过以下四种方式引用另一个代码段：

- 目标操作数含有目标代码段的段选择子。
- 目标操作数指向一个调用门的描述符，该描述符含有目标代码段的段选择子。
- 目标操作数指向一个 TSS，该 TSS 保护目标代码段的段选择子。
- 目标操作数指向一个任务门，这个任务门指向一个 TSS，这个 TSS 含有目标代

码段的段选择子。

下面几节描述了前两种引用方式。通过任务门或 TSS 进行进程控制转移的信息参看 6.3.“任务切换”。

SYSENTER 和 SYSEXIT 指令是特殊指令，用来快速调用操作系统例程以及从例程中返回。这些指令将在 4.8.7.“通过 SYSENTER 和 SYSEXIT 指令快速调用系统例程”中作简短的讨论。

4.8.1.直接调用或者跳转到代码段

JMP、CALL 和 RET 指令的近的指令形式只是在当前代码段内进行进程控制的转移，所以不进行特权级的检验。而这些指令的远距离的指令形式将控制转移到另外一个代码段，所以处理器要进行特权级检验。

当不通过调用门转移进程控制到另外一个代码段时，处理器检查以下四种特权级和类型信息（见图 4-5）：

- CPL（这里，CPL 是调用代码段的特权级，也就是发起调用或跳转的例程所在的代码段。）

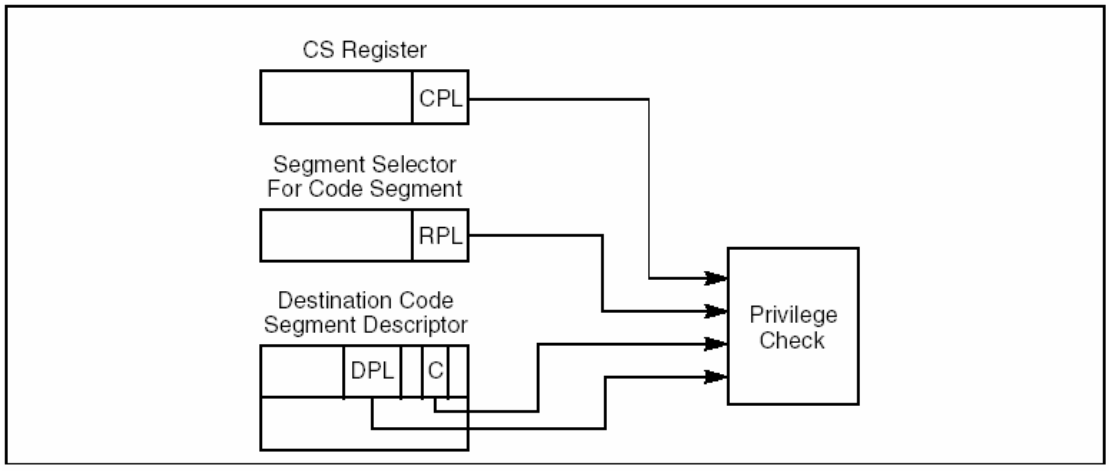

Figure 4-5. Privilege Check for Control Transfer Without Using a Gate

- 被调例程所在目标代码段的描述符的 DPL。
- 目标代码段的段选择子的 RPL。
- 目标代码段描述符的一致性（C）标志，这个标志确定一个段是一个一致性代码段（标志为 1）还是非一致性代码段（标志为 0）。（更多关于这个标志的信息参看 3.4.3.1.“代码段和数据段描述符类型”。）

处理器检验 CPL、RPL 和 DPL 的规则取决于 C 标志的设置，这个在下面章节中描述。

4.8.1.1. 访问非一致性代码段

当访问非一致性代码段时，调用例程的 CPL 必须与目标代码段的 DPL 相等，否则处理器会产生一个一般保护异常（#GP）。

比如，在图 4-6 中，代码段 C 是一个非一致性代码段。那么，代码段 A 中的例程可以调用代码段 C 中的例程（使用段选择子 C1），因为它们有相同的特权级（代码段 A 的 CPL 与代码段 C 的 DPL 相等）。然而，代码段 B 中的例程（使用段选择子 C2 或 C1）就不能调用代码段 C 中的例程，因为这两个代码段有不同的特权级。

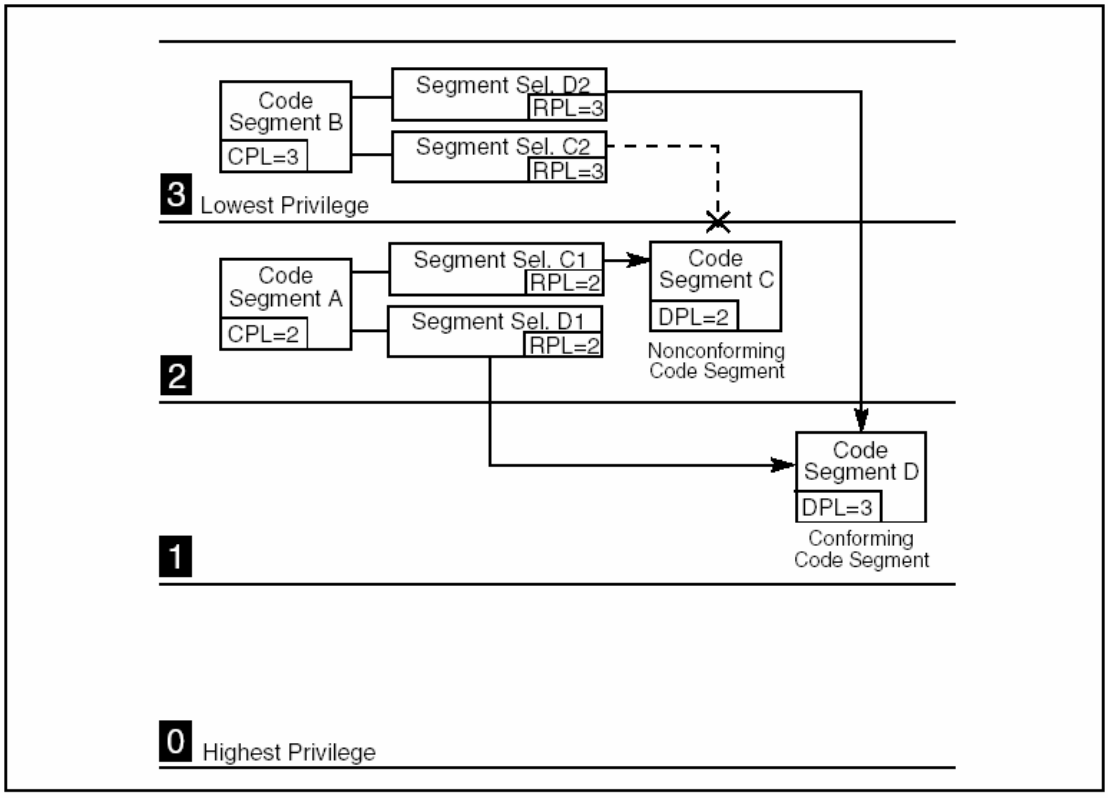

Figure 4-6. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

指向非一致性代码段的段描述符的 RPL 对特权级检验的影响是有限的。RPL 必须在数值上小于或者等于调用例程的 CPL，才能成功地进行控制转移。所以，在图 4-6 的例子中，段选择子 C1、C2 的 RPL 可以被合法的设置为 0、1 或者 2，但是不能是 3。

当 CS 寄存器装入一个非一致性代码段的段选择子时，不改变特权级域的值。这意味着，调用例程的 CPL 被保留。即使该段选择子的 RPL 与 CPL 不一样时，也是这样。

4.8.1.2. 访问一致性代码段

当访问一致性代码段时，调用例程的 CPL 可以在数值上等于或大于（较低特权级）

目标代码段的 DPL。仅当 CPL 小于 DPL 的时候，处理器产生一个一般保护异常（#GP）。（当目标代码段是一致性代码段时，不用检验目标代码段的 RPL。）

在图 4-6 中的例子中，代码段 D 是一个一致性代码段。因此，代码段 A 和代码段 B 中的调用例程都可以访问代码段 D（分别使用段选择子 D1 或 D2），因为它们的 CPL 都大于或者等于这个一致性代码段的 DPL。（此处与图 4-6 不符，似乎有错误——译注。）**对于一致性代码段而言，DPL 代表调用例程对该代码段进行成功访问时所拥有的特权级的最低数值。**

（注意，段选择子 D1 和 D2 除了 RPL 不一样外，其余的都是一样的。但是由于访问一致性代码段时不检验 RPL，因此，在这种情况下，这两个段选择子本质上是一样的。）

当进程控制转入一个一致性代码段时，即使目标代码段的 DPL 小于 CPL，也不改变 CPL（CS 寄存器中的特权级域不变。即装载新代码段的选择子到 CS 中时，CPL 域保持不变——译者注）。只有在这种情况下，CPL 可以与当前代码段的 DPL 不同。同时，因为 CPL 没有变化，栈也不用切换。

一致性代码段主要用于数学函数库和异常处理程序的代码模块，这些代码对应用程序提供支持，但是并不访问受保护的系统设施。这些模块是操作系统或者管理程序的一部分，但是它们可以以更低特权级来执行。当进程切换到一个一致性代码段时，保留当前进程的 CPL 可以防止应用程序在一致性代码段的特权级上访问非一致性代码段（它的 DPL 与当前进程的 CPL 一致），因而防止它访问更高特权级的数据。

多数代码段是非一致性的。对这些段而言，除非控制转移是通过一个调用门完成的，否则，进程控制只可以转到相同特权级的代码段上。这个在下面的章节中描述。

4.8.2. 门描述符

为了对不同特权级的代码段进行可以控制的访问，处理器提供了一组特殊的描述符，叫做门描述符。有四种门描述符：

- 调用门。
- 陷阱门。
- 中断门。
- 任务门。

任务门用于任务切换，将在第 6 章“任务管理”中讨论。陷阱门和中断门是特别

的调用门，用来调用异常和中断处理程序，将在第五章“中断和异常处理”中有描述。本章只关注调用门。

4.8.3.调用门

调用门为不同特权级间的进程控制转移提供了便利。它们一般只用在使用特权级保护机制的操作系统或者管理程序中。调用门也可用于 16 位和 32 位代码段之间的进程控制转移，这在 17.4.“混合尺寸代码段之间的控制转移”中进行描述。

图 4-7 显示了一个调用门描述符的格式。调用门描述符可以在 GDT 或 LDT 中，但是不能在中断描述符表（IDT）中。它具有六个方面的功能：

- 确定将要访问的代码段。
- 定义例程在指定代码段中的入口。
- 指定调用例程的所应当有的特权级。

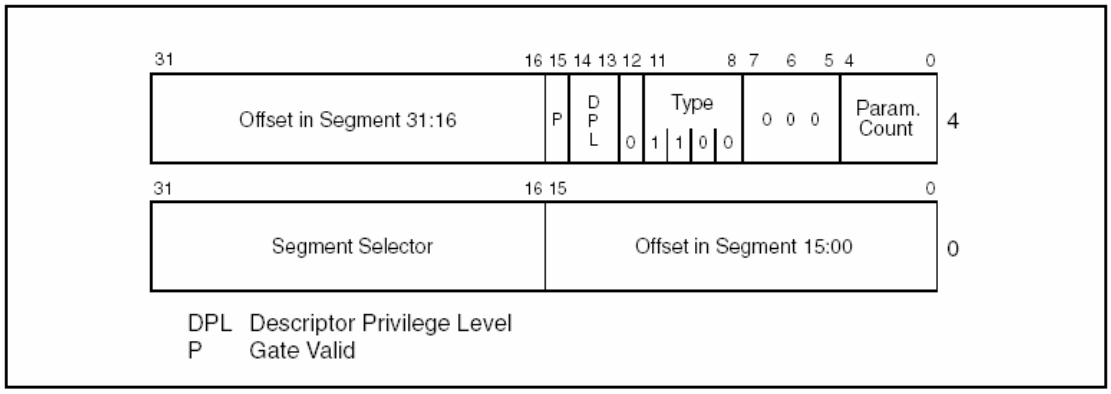

Figure 4-7. Call-Gate Descriptor

- 如果有栈切换，就确定在栈之间拷贝的可选参数的个数。
- 定义压入目标栈的值的尺寸：16 位的门执行 16 位的压栈，32 位的门执行 32 位的压栈。
- 确定调用门描述符是否有效。

调用门中的段选择子域确定将要访问的代码段。偏移域确定代码段中的入口点。这个入口点通常是指定例程的第一条指令。DPL 域指明调用门的特权级，也就是通过该调用门访问例程所必须具备的特权级。P 标志指明调用门描述符是否有效。（调用门指向的代码段是否在内存由代码段描述符中的 P 标志确定）。参数计数域的作用是，当发生栈切换时，需要从调用进程的栈拷贝到新栈的参数个数（见 4.8.5.“栈切换”）。对于 16 位调用门来说，参数计数域确定需要拷贝的字数，对于 32 位调用门来说，就是

需要拷贝的双字数。

注意，门描述符中的 P 标志通常总是被置为 1。如果它被置为 0，当进程试图访问该描述符时，将产生一个不存在异常（＃NP）。操作系统可以为某个特定的意图使用这个 P 标志。比如说，可以使用 P 标志来跟踪这个门被使用的次数。这时，P 标志通常最初被置为 0 以进入不存在异常处理程序。该异常处理程序就将一个计数器加 1 并且将 P 标志置 1。从这个异常处理程序返回后，门描述符就是有效的了。

4.8.4.通过调用门访问代码段

为了访问调用门，CALL 或 JMP 指令的目标操作数中必须有一个指向门的远指针。远指针的段选择子标识调用门（见图 4-8）；指针里还要有偏移，只是处理器不使用也不检验。（这个偏移可以被置为任意值。）

一旦处理器访问调用门，它就使用调用门中的段选择子来定位目标代码段的段描述符。（这个段描述符在 GDT 中或者 LDT 中。）之后，把描述符中的段基地址和调用门中的偏移相加，组成一个线性地址，这就是被调例程在代码段中入口点的线性地址。

如图 4-9 中所示，通过调用门进行的进程控制转移的有效性检验用到了四种不同的特权级：

- CPL（当前特权级）。
- 调用门选择子的 RPL。
- 调用门描述符的 DPL。
- 目标代码段的段描述符的 DPL。

同时还要检验目标代码段的段描述符的一致性标志（C）标志。

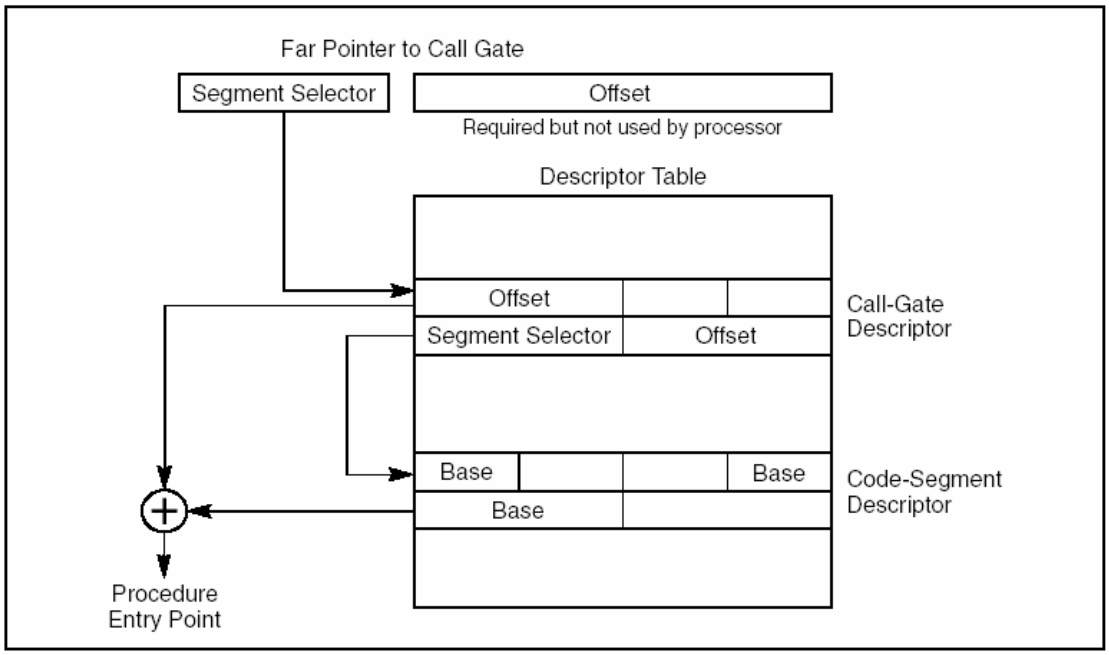

Figure 4-8. Call-Gate Mechanism

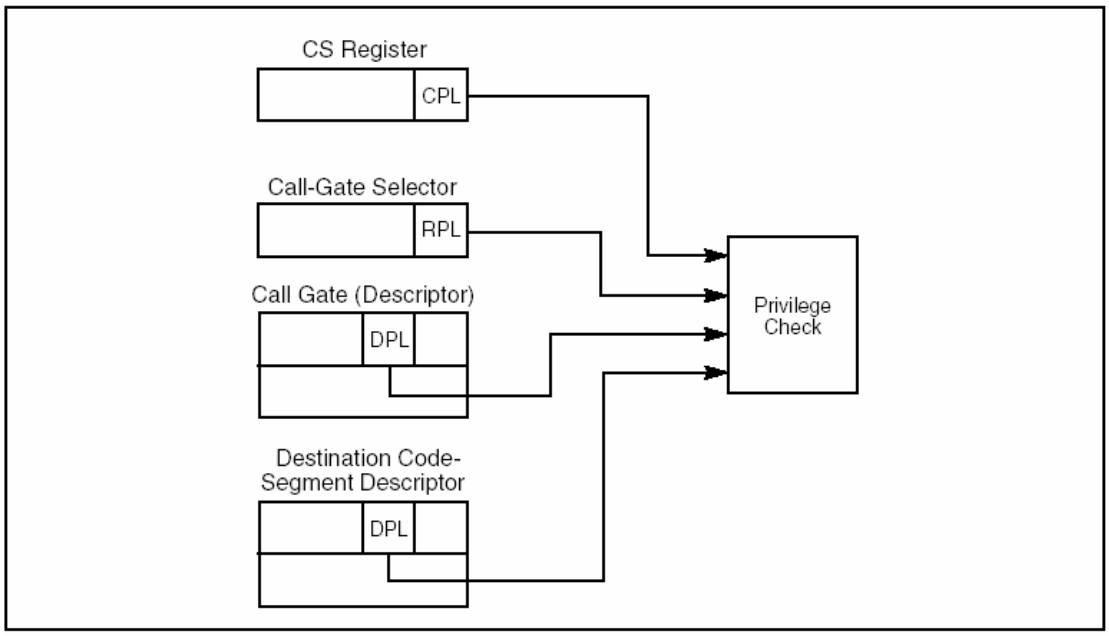

Figure 4-9. Privilege Check for Control Transfer with Call Gate

如表 4-1 所示，CALL 指令和 JMP 指令发起的进程控制转移的特权检验规则是不同的。

表 4-1 调用门的特权检验规则

指　令	特权检验规则
CALL	CPL <= 调用门 DPL；RPL <= 调用门 DPL 目标一致性代码段 DPL <= CPL 目标非一致性代码段 DPL <= CPL

JMP	CPL <= 调用门 DPL；RPL <= 调用门 DPL 目标一致性代码段 DPL <= CPL 目标非一致性代码段 DPL = CPL

调用门描述符的 DPL 域指定调用进程访问调用门应当具有的特权级的最大数值。也就是，要访问调用门，调用进程的 CPL 必须等于或小于调用门的 DPL。比如，在图 4-10 中，调用门 A 的 DPL 是 3。所以任何 CPL（特权级 0 到 3）的调用进程都可以访问这个调用门，包括代码段 A、B 和 C 中的调用进程。调用门 B 的 DPL 为 2，所以只有 CPL 为 0、1、2 的调用进程可以访问调用门 B，包括代码段 B 和 C 中的调用进程。点划线显示，在代码段 A 中的调用进程不能访问调用门 B。

调用门段选择子的 RPL 必须和调用例程的 CPL 一样满足同样的条件，也就是 RPL 必须小于或等于调用门的 DPL。在图 4-10，代码段 C 中的调用进程可以使用门选择子 B2 或 B1 访问调用门 B，但是不能使用门选择子 B3 来访问调用门 B。

如果调用进程和调用门之间的特权级检验通过了，处理器紧接着就检验代码段描述符 DPL 和调用进程的 CPL。此处，CALL 指令和 JMP 指令的特权级检验的规则是不同的。只有 CALL 指令可以使用调用门将进程控制转移到一个特权级更高的非一致性性代码段。也就是说，可以访问一个 DPL 小于 CPL 的非一致性代码段。JMP 指令仅能使用调用门将进程控制转移到一个 DPL 等于 CPL 的非一致性代码段。CALL 和 JMP 指令都可以将进程控制转移到一个特权级更高的一致性代码段；也就是，转移到一个 DPL 小于或等于 CPL 的一致性代码段。

如果调用特权级更高的非一致性目标代码段，CPL 就降为目标代码段的 DPL 特权级，并且会发生栈切换（见 4.8.5.“栈切换”）。如果调用或者跳转到一个特权级更高的一致目标代码段，CPL 不会发生变化，也不会发生栈切换。

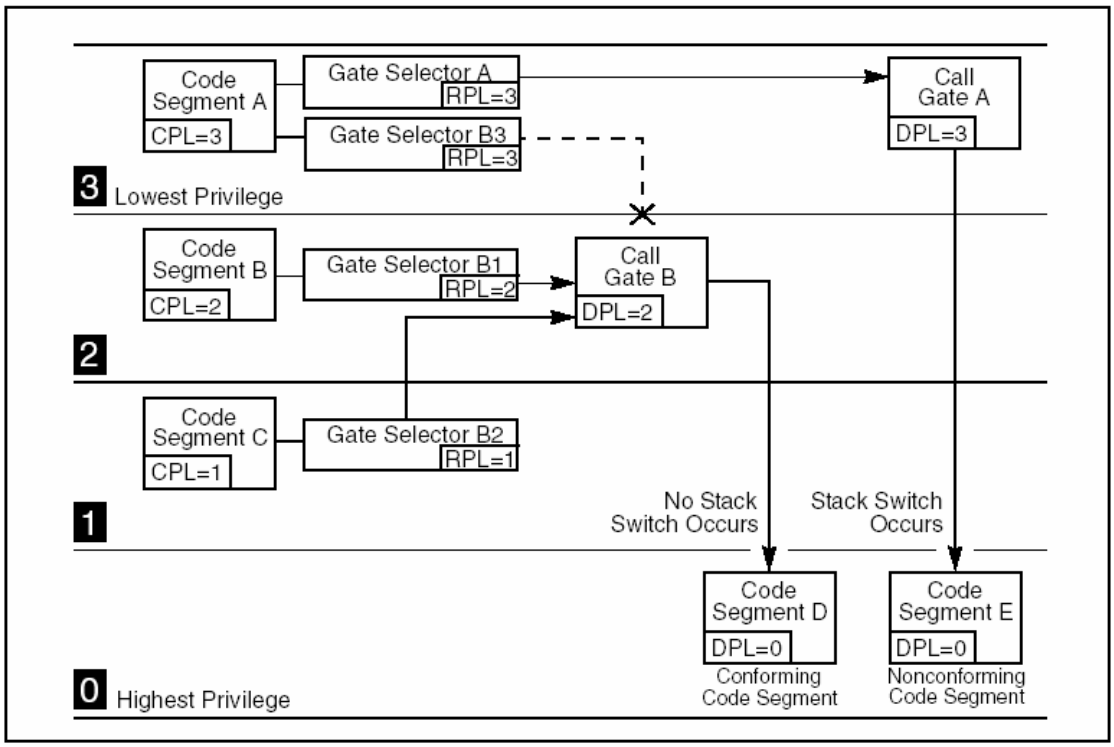

Figure 4-10. Example of Accessing Call Gates At Various Privilege Levels

调用门允许单个代码段上的有被不同特权级进程访问的例程。比如，放置在某个代码段中的操作系统有一些服务程序，既给操作系统也给应用软件（比如字符 I/O 处理例程）使用。为这些例程建立的调用门可以拥有任何特权级（0 到 3）。可以为那些只供操作系统使用的服务(比如初始化设备驱动程序的例程)建立特权级更高的调用门（DPL 为 0 或 1）。

4.8.5.栈切换

无论何时使用调用门来转移进程控制到一个特权级更高的非一致性代码段（也就是非一致性目标代码段的 DPL 小于 CPL），处理器都会自动地切换栈到目标代码段相应特权级的栈上。进行这样的栈切换主要是为了防止特权级更高的例程因为栈空间的不足而崩溃。同时也防止特权级较低的进程通过共享栈干扰(有意或无意地)高特权级的进程。

每个任务都必须定义最多 4 个栈：一个应用代码（特权级 3）栈，特权级 2、1、0 代码的栈各一个。(如果仅使用 2 级特权：3 和 0，就只定义 2 个栈。）这些栈每个都必须放在一个单独的段中，用一个段选择子和段中的偏移（栈指针）来指定。

当执行特权级 3 的代码时，特权级 3 的栈的段选择子和栈指针分别放在 SS 和 ESP

寄存器中；在栈切换时，被自动保存在被调进程的栈中。

特权级为 0、1 和 2 的栈的指针存在当前运行任务的 TSS 中（见图 6-2）。这些指针由一个段选择子和一个栈指针（载入在 ESP 寄存器中）组成。这些指针的初始值都是严格的只读的。任务运行期间，处理器不改变它们的值。它们只是用来在调用更高特权级时创建新的栈。这些栈在从被调例程返回时被释放。下次同一例程被调用时，还使用这些栈指针的初始值创建新的栈。（TSS 并不为特权级 3 的进程指定栈，因为处理器不允许进程控制从 CPL 为 0、1、2 的运行例程转移到 CPL 为 3 的运行例程，除非是返回过去。）

操作系统负责为用到的所有特权级创建栈和栈段描述符，并负责将这些栈指针的初始值载入 TSS 中。每个栈都必须是可读写的（在它的段描述符中的类型域定义），而且必须有足够的空间（在界限域中定义）来容纳如下数据：

- 调用例程的 SS、ESP、CS 和 EIP 等寄存器的内容。
- 被调例程所需的参数和临时变量。
- EFLAGS 寄存器和出错码，当隐含调用异常或中断处理函数时。

栈需要有足够的空间来包含许多套上述数据项，因为例程经常会调用其它例程，而且操作系统可能会支持多重中断的嵌套。每个栈必须足够大，以顾及在它的特权级内最糟糕的嵌套情况。

（如果操作系统不使用处理器的多任务机制，依然至少必须创建一个 TSS，以满足栈相关的目标。）

当某个通过调用门的例程调用导致了特权级的改变，处理器执行如下步骤进行栈切换，并且在新的特权级上执行被调例程：

1．使用目标代码段的 DPL（也就是新的 CPL）从 TSS 中选择一个指向新栈的指针（段选择子和栈指针）。

2．从当前 TSS 中读取将要切换到的栈的段选择子和栈指针。在读栈段选择子、栈指针或栈段描述符时，如果检查到任何界限违例，都会产生一个非法 TSS 异常（＃TS）。

3．检验栈段描述符的特权级和类型，如果发现违例，就会产生一个非法 TSS 异常（＃TS）。

4．暂时保存当前 SS 寄存器和 ESP 寄存器的值。

5．将新栈的段选择子和栈指针载入 SS 寄存器和 ESP 寄存器。

6.将暂时保存的 SS 寄存器和 ESP 寄存器的值(调用例程的)压入新栈(见图 4-11)。

7．将调用门的参数计数域所指定个数的参数从调用例程的栈拷贝到新栈。如果参数个数域为 0，则一个参数也不拷贝。

8．将返回指令指针（当前 CS 寄存器和 EIP 寄存器）压入新栈。

9．从调用门中将新代码段的段选择子和新指令指针分别载入 CS 寄存器和 EIP 寄存器，然后开始执行被调例程。

有关通过调用门进行远调用时的特权级检验和其它保护性检验的相关信息，参看《第 2 卷：指令集参考》第三章中的 CALL 指令的描述。

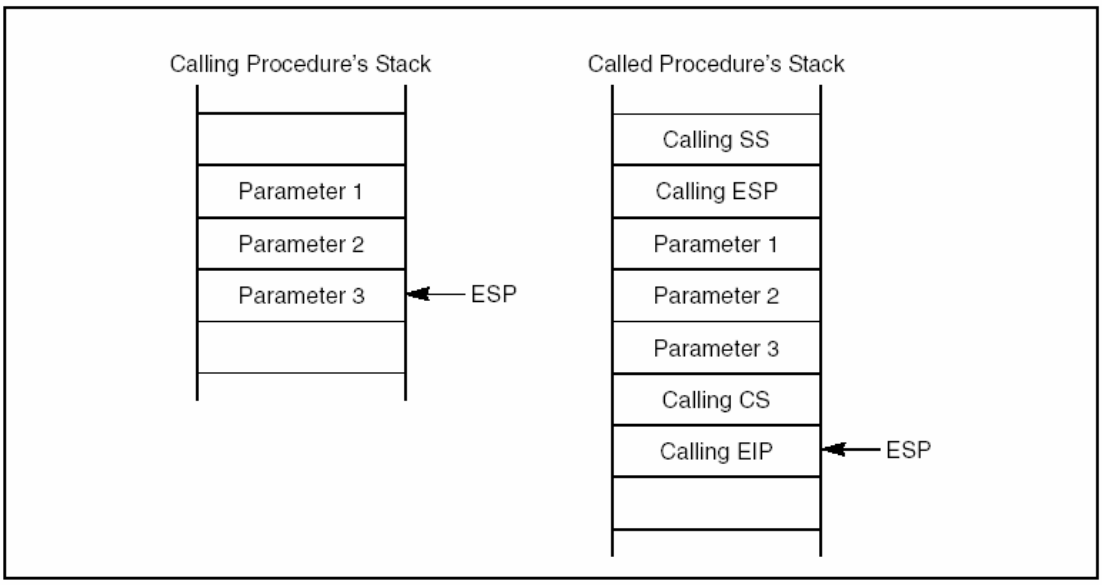

Figure 4-11. Stack Switching During an Interprivilege-Level Call

调用门的参数计数域指定了处理器需要从调用例程的栈拷贝到被调例程的栈的数据项个数（最多 31 个）。如果有超过 31 个数据项要传给被调例程，那么，其中的一个参数可以是指向一个数据结构的指针，或者使用已保存好的 SS 寄存器和 ESP 寄存器的值来访问在旧栈空间的参数。如 4.8.3.“调用门”中所述，传给被调例程的数据项的长度取决于调用门的长度。

4.8.6.从被调例程返回

RET 指令可以用来执行一个近返回、同一个特权级的远返回和不同特权级的远返回。这个指令用来执行从一个由 CALL 指令调用的例程返回。它不支持从一个 JMP 指令的返回，因为 JMP 指令并不在栈上保存返回指令的指针。

近返回只是在当前代码段内进行控制转移，因而，处理器仅仅执行界限检验。当处理器将返回的指令指针弹入 EIP 寄存器时，它会检验指针是否超出了当前代码段的

界限。

在同一特权级的远返回时，处理器弹出将返回到的代码段的段选择子，同时还从栈上弹出返回的指令指针。通常，这些指针都是有效的，因为它们都是被 CALL 指令压入栈中的。然而，处理器仍然会进行特权级的检验来检测当前例程是否改变了指针或者是否对栈维护不当。

要求特权级改变的远返回，只能返回到较低的特权级（也就是，返回到的代码段的 DPL 在数值上大于 CPL)。处理器使用为调用例程保存的 CS 寄存器的值中的 RPL 域(参加图 4-11）来确定是否可以返回到数值上较大的特权级。如果 RPL（特权低）的数值大于 CPL，就执行跨特权级返回。

当执行远返回到调用例程时，处理器进行如下步骤的操作（关于返回前、后栈中的内容变化的演示，参看《第 1 卷：基础架构》中的图 6-2 和图 6-4)：

1．检查保存的 CS 寄存器的 RPL 域来确定返回时是否需要改变特权级。

2．从被调例程的栈上载入 CS 寄存器和 EIP 寄存器。(类型和特权级的检验分别在代码段描述符和代码段选择子的 RPL 上进行的)。

3.（如果 RET 指令包含一个参数计数操作数，并且返回要求改变特权级。）把参数计数（从 RET 指令获得，以字节计）加进当前 ESP 寄存器的值（在 CS 寄存器和 EIP 寄存器的值弹出之后)，以在被调例程的栈中跳过参数区。结果，ESP 寄存器中的值就指向了保存的调用例程栈的 SS 寄存器和 ESP 寄存器的值。(注意，RET 指令中的字节计数必须与调用例程引用的调用门中的参数计数相匹配，它在实施最初调用的时候是乘以参数的长度的。)

4.（如果返回要求改变特权级。）将保存的 SS 和 ESP 值载入 SS 寄存器和 ESP 寄存器，并切回到调用例程的栈。被调例程的栈的 ESP 和 SS 值就被丢弃了。载入栈段选择子或栈指针时，检测到任何界限违例都会产生一个一般保护异常（#GP)。对新栈段描述符也要进行类型和特权级违例的检验。

5.（如果 RET 指令含有一个参数计数的操作数。）将参数计数（从 RET 指令获得，以字节计）加进 ESP 寄存器的值，跳过在调用例程栈上的参数。此时并不检验结果 ESP 的值是否超过了栈段的界限。如果 ESP 的值超过了段界限，要直到下一次栈操作才能被发现。

6.（如果该返回需要改变特权级。）检验 DS、ES、FS 和 GS 段寄存器的内容。如果这些寄存器中的某一个所指向的段的 DPL 小于新的 CPL（不包括一致性代码段)，该寄

存器就会被赋予一个空的段选择子。

有关在远返回时，处理器进行特权级检验和其它保护性检验的更详细描述，参看《第2卷：指令集参考》第三章中关于RET指令的描述。

4.8.7.使用SYSENTER和SYSEXIT指令快速调用系统例程

SYSENTER和SYSEXIT指令是在奔腾2处理器时引入IA-32架构的，目的是提供一种调用操作系统或者管理程序的快速机制（低开销）。SYSENTER指令用于运行在特权级3上的用户代码访问操作系统或者管理程序的例程。SYSEXIT例程用于特权级为0的操作系统或者管理程序例程能够快速返回特权级为3的用户代码。SYSENTER指令可以在特权级3、2、1上执行；SYSEXIT指令只能在特权级0上执行。

SYSENTER和SYSEXIT指令是伙伴指令，但是它们并不构成“调用/返回”对，因为SYSENTER指令并不保存任何供SYSEXIT指令在返回时使用的状态信息。

这两个指令的目标指令和栈指针不是通过指令操作数来指定的。相反，它们通过放置在MSR寄存器和几个通用寄存器中的参数来确定的。对于SYSENTER指令而言，处理器从下述几个来源获取在特权级0上的目标指令和栈指针：

- 目标代码段——从SYSENTER_CS_MSR中读取。
- 目标指令——从SYSENTER_EIP_MSR中读取。
- 栈段——把SYSENTER_CS_MSR的值加上8获得。
- 栈指针——从SYSENTER_ESP_MSR中读取。

对于SYSEXIT指令而言，特权级为3的目标指令和栈指针是通过如下方式来确定的：

- 目标代码段——将SYSENTER_CS_MSR中的值加上16获得。
- 目标指令——从EDX寄存器中读取。
- 堆栈段——将SYSENTER_CS_MSR中的值加上24获得。
- 栈指针——从ECX寄存器中读取。

SYSENTER和SYSEXIT指令执行“快速的”调用和返回，因为当执行SYSENTER指令时，它们强迫处理器进入一个预定的特权级0状态；当执行SYSEXIT指令时，进入预定的特权级3状态。通过强制预定的和一致性的处理器状态，大大减少了通常在远调用进入另外一个特权级时的特权级检验。同时，通过将目标场境存放在MSR寄存器和

通用寄存器中来避免访问内存，除非要取目标代码。

如果还有什么状态需要保存以便返回到调用例程，就需要显式地由调用例程保存或者通过编程惯例事先约定。

4.9.特权指令

某些系统指令（称为“特权指令”）是禁止应用程序使用的。这些特权指令控制着系统功能（比如装载系统寄存器）。仅当 CPL 为 0 时才能执行它们。如果 CPL 不为 0 时执行它们，将会产生一个一般保护异常（#GP）。以下系统指令就是特权指令：

- LGDT——装载 GDT 寄存器。
- LLDT——装载 LDT 寄存器。
- LTR——装载任务寄存器。
- LIDT——装载 IDT 寄存器。
- MOV（控制寄存器）——装载和存储控制寄存器。
- LMSW——装载机器状态字。
- CLTS——清除寄存器 CR0 中的任务切换标志。
- MOV（调试寄存器）——装载和存储调试寄存器。
- INVD——使缓存失效，无写回。
- WBINVD——使缓存失效，有写回。
- INVLPG——使 TLB 项失效。
- HLT——使处理器停止。
- RDMSR——读模型相关寄存器。
- WRMSR——写模型相关寄存器。
- RDPMC——读取性能监视计数器。
- RDTSC——读时间戳计数器。

某些特权指令只是在最近的 IA-32 处理器家族中才出现的（见 18.10.“Pentium 及其以后的 IA-32 处理器新指令”）。寄存器 CR4 中的 PCE 和 TSD 标志（分别是第 4 位和第 2 位）分别开启 RDPMC 和 RDTSC 指令在任何 CPL 下执行。

4.10.指针验证

在保护模式下运行时，处理器会验证所有的指针，强迫段之间的保护，维护特权级之间的冲突。指针验证包括如下检验：

1．检验访问权限以确定段类型与它的使用是否兼容。

2．检验读写权限。

3．检验指针偏移是否超出了段界限。

4．检验是否指针的提供者访问该段。

5．检验偏移的对齐。

在指令执行期间，处理器自动执行第一、二、三步检验。软件必须显式地发出 ARPL 指令来请求第四步检验。在特权级 3 上如果对齐检验开启的话，则第五步检验（偏移检验）自动进行。

4.10.1.检验访问权限（LAR 指令）

当处理器通过一个远指针来访问某个段时，它要对该远指针所指向的段描述符进行访问权限检验。这个检验就是要确定段描述符的类型和 DPL 与将要进行的操作是否兼容。比如，在保护模式下进行一个远调用时，段描述符的类型必须是指向一个一致或非一致性代码段、调用门、任务门或者 TSS。而且，如果调用的是一个非一致性代码段，则该代码段的 DPL 必须等于 CPL，代码段选择子的 RPL 必须小于或等于 DPL。如果发现类型或者特权级不兼容，就产生相应的异常。

为了防止类型不兼容异常的产生，程序可以使用 LAR（装载访问权限）指令来检验段描述符的访问权限。LAR 指令指定被检验访问权限的段描述符的段选择子和一个目的寄存器，然后执行如下操作：

1．检验段选择子是否非空。

2．检验段选择子指向的段描述符是否在描述符表（LDT 或 GDT）的界限内。

3．检验段描述符的类型是一个代码段、数据段、LDT 段、调用门、任务门还是 TSS。

4．如果段是一个非一致性代码段，则检验该段描述符在当前 CPL 下是否可见（也就是，CPL 和段选择子的 RPL 是否小于或等于 DPL）。

5．如果特权级和类型检验都通过了，就将段描述符的第二个双字载入到目的寄存

器（用数值 00FXFF00H 屏蔽，这里 X 指的是相应的 4 位未定义），并且设置 EFLAGS 寄存器中的 ZF 标志。如果在当前的特权级上，段选择子是不可见的或者对 LAR 指令而言是无效的，就不改变目的寄存器并清除 ZF 标志。

一旦装载了目的寄存器，软件就可以进行访问权限的其它检验。

4.10.2.检验读写权限（VERR 和 VERW 指令）

当处理器访问代码或数据段时，它首先检验分配给这个段的读写权限，确认试图将要进行的读写操作是否被允许。程序可以使用 VERR（验证读）和 VERW（验证写）指令来检验读写权限。这两个指令都指定被检验段的段选择子。它们执行如下操作：

1．检验段选择子是否为空。

2．检验段选择子指向的段描述符是否在描述符表（GDT 或 LDT）的界限内。

3．检验段描述符的类型是代码段还是数据段。

4．如果该段是一个非一致性代码段，则检验它的描述符在当前的 CPL 下是否可见（也就是 CPL 和段选择子的 RPL 是否小于或等于 DPL）。

5．检验该段是否可读（对 VERR 指令而言）或可写（对 VERW 指令而言）。

如果段在当前 CPL 下是可见和可读的，则 VERR 指令设置 EFLAGS 寄存器中的 ZF 标志；如果段是可见和可写的，则 VERW 指令设置 ZF 标志。（代码段永远不可写。）当然，如果这些检验中有失败的，则 ZF 标志就会被清零。

4.10.3.检验指针偏移是否在段界限内（LSL 指令）

当处理器访问一个段时，要进行界限的检验以确保偏移在段界限内。软件可以使用 LSL（装载段界限）指令来进行界限的检验。与 LAR 指令一样，LSL 指令指定被检验界限的段描述符的段选择子和一个目标寄存器。紧接着指令执行如下操作：

1．检验段选择子是否为空。

2．检验段选择子所指的段描述符是否在描述符表（GDT 或 LDT）的界限内。

3．检验段描述符的类型是代码段、数据段、LDT 还是 TSS。

4．如果段不是一个一致性代码段，则检验段描述符在当前 CPL 下是否可见（也就是，CPL 和段选择子的 RPL 是否小于或等于 DPL）。

5．如果特权级和类型检验都通过了，将整理出的界限（界限的增长的尺寸随着段

描述符的G标志设置而定的)载入目标寄存器并置位EFLAGS寄存器中的ZF标志。如果段选择子在当前特权级下不可见或者对 LSL 指令来说是一个无效类型的段选择子，则指令不修改目标寄存器，并将ZF标志清零。

一旦界限被装入目标寄存器，软件就可以将段界限与指针中的偏移进行比较了。

4.10.4.检验调用者的访问权限（ARPL指令）

段选择子的RPL域是用于将调用例程的特权级（调用例程的CPL）带到被调例程。被调例程使用RPL来确定是否允许访问某个段。通俗地说,RPL将被调例程的特权级“弱化”至RPL。

典型用法是操作系统例程使用 RPL 来阻止低特权的应用程序访问高特权段内的数据。当一个操作系统例程（被调例程）从应用程序（调用例程）收到一个段选择子时，它将段选择子的 RPL 设置为调用例程的特权级。接着，当操作系统使用这个段选择子访问与它自己相关的段时，处理器使用调用例程的特权级（保存在RPL中），而不是数值上更低的操作系统例程的特权级，来进行特权级检验。因而，RPL确保操作系统不代表应用程序访问段，除非应用程序本身具有对这个段的访问权限。

图4-12给出了一个处理器如何使用RPL域的例子。在这个例子中，一个应用程序（位于代码段 A 中）持有一个指向特权级更高的数据结构（位于特权级 0 的数据段 D 中）的段选择子（段选择子D1）。

应用程序不能访问数据段D，因为它没有足够的特权，但是操作系统（在代码段C中）可以。所以为了访问数据段D，应用程序执行一个对操作系统的调用，将段选择子D1放在栈中作为参数传给操作系统。在传这个段选择子前，应用程序（一直执行的很好）将段选择子中的RPL设置为当前特权级（在这个例子中是3）。如果操作系统试图使用段选择子D1来访问数据段D，则处理器会比较CPL（调用之后是0）、段选择子D1的RPL以及数据段D的DPL（这里是0）。由于RPL大于DPL，所以对数据段D的访问被拒绝。这样，处理器的保护机制阻止了操作系统对数据段 D 的访问，因为应用程序的特权级值（段选择子B中的RPL表示）大于数据段D的DPL。

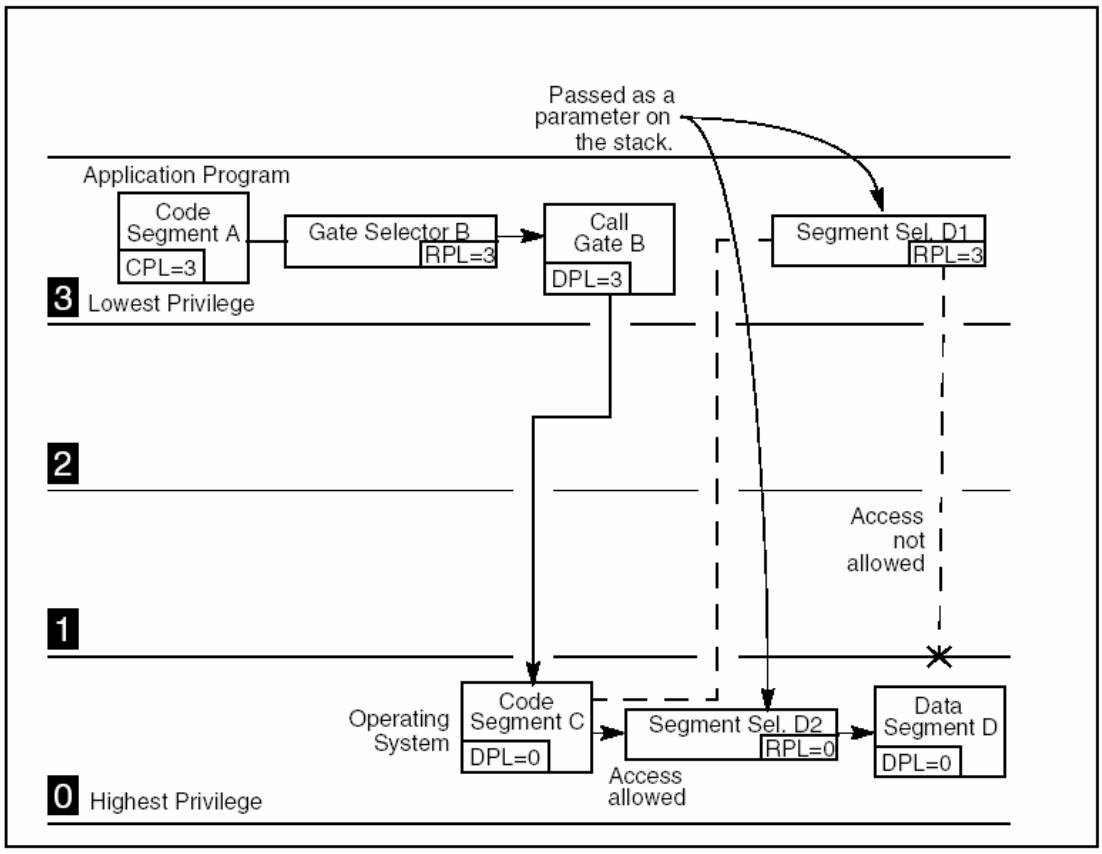

Figure 4-12. Use of RPL to Weaken Privilege Level of Called Procedure

现在假设应用程序不是将段选择子的 RPL 设置为 3，而是设置为 0（段选择子 D2）。现在操作系统可以访问数据段 D，因为它的 CPL 和段选择子 D2 的 RPL 都与数据段 D 的 DPL 相等。

因为应用程序可以将段选择子的 RPL 设置为任何值，因此它就有可能使用一个数值较低的特权级的例程去访问受保护的数据结构。这种降低段选择子 RPL 的能力突破了处理器的保护机制。

因为被调例程不能依赖调用例程来正确的设置 RPL，操作系统例程（运行在数值较低的特权级）从数值较高的特权级的例程收到段选择子以后，需要测试一下段选择子的 RPL 来确定特权级是否合适。ARPL（调整请求特权级）指令就是出于这个目的。这个指令调整一个段选择子的 RPL 来匹配另外一个段选择子的 RPL。

图 4-12 中的例子演示了如何使用 ARPL 指令。当操作系统从应用程序接收到段选择子 D2 时，它使用 ARPL 指令来比较该段选择子的 RPL 和应用程序的特权级（在栈中的代码段选择子中）。如果 RPL 小于应用程序的特权级，则 ARPL 指令就改变段选择子的 RPL 来匹配应用程序的特权级（段选择子 D1）。因而，使用这个指令可以阻止数值更高的特权级的例程通过降低段选择子的 RPL 来访问数值更低的特权级（更高特权）的

段。

注意，应用程序的特权级可以通过应用程序代码段的段选择子的 RPL 域来确定。这个段选择子存放在栈中作为对操作系统调用的一部分。操作系统可以从栈中复制这个段选择子到一个寄存器中，作为 ARPL 的一个操作数。

4.10.5.对齐检验

当 CPL 为 3 时，内存引用的对齐是通过设置 CR0 寄存器中的 AM 标志和 EFLAGS 寄存器中的 AC 标志来检验的。没有对齐的内存应用会产生一个对齐异常（＃AC)。当特权级为 0、1、2 时，处理器不会产生对齐异常。表 5-7 描述了当开启对齐检验时的对齐要求。

4.11.页级保护

页级保护可以单独应用，也可以应用在段上。当在平坦内存模型下使用页级保护时，可以将管理态的代码和数据（操作系统或管理程序）与用户态的代码和数据（应用程序）分开。还可以将有代码的页设置写保护。当段级保护和页级保护机制结合起来的时候，页级读写保护允许段内有更多的保护粒度。

每次内存引用检验是和页级保护（段级保护也是）检验一起进行的，以确保保护检验得以满足。所有的检验都在内存周期开始之前完成的，任何违例都会阻止内存周期启动并导致内存页异常（页故障异常)。因为这些检验与地址转换同时进行，不会有任何性能损失。

处理器进行 2 个页级保护的检验：

- 可寻址区域的限制（管理态和用户态)。
- 页类型（只读或可读可写)。

在检验过程中有任何违例都会导致页故障异常。有关页故障异常机制的详细解释，参见第五章“中断 14——页故障异常（＃PF)”。本章描述保护违例导致的页故障异常。

4.11.1.页保护标志

页的保护信息包含在页目录表项或页表项两个标志中(见表 3-14)：读/写标志(第

1 位）和用户/管理标志（第 2 位）。第一级和第二级的页表（也就是，页目录表和页表）都应用了保护检验。

4.11.2.限定可寻址区间

页级保护机制允许基于 2 个特权级对页的访问进行限定：

- 管理态（U/S 标志为 0）——（最高特权级）用于操作系统或管理程序、其它系统软件（比如设备驱动程序）以及受保护的系统数据（比如页表）。
- 用户态（U/S 标志为 1）——（最低特权级）用于应用程序代码和数据。

段特权级到页特权级的映射方式如下。如果处理器当前运行在 0、1、2 的 CPL 上，它就是处在管理态；如果运行在 CPL 为 3 上，它就处在用户态。当处理器在管理态时，它可以访问所有的页；当处于用户态时，它只能访问用户级的页。(注意，控制寄存器中的 CR0 的 WP 标志会改变管理态许可，见 4.11.3. “页类型”的描述。)

注意，要使用页级保护机制，代码段和数据段必须建立至少 2 级基于段的特权级：0 级用于管理态的代码段和数据段，3 级用于用户态的代码段和数据段。(这种模型下，栈放置在数据段中)。为了减少对段的使用，可以使用平坦内存模型（见 3.2.1.“基本平坦模型”)。

这里，用户态和管理态的代码段、数据段都是从线性地址 0 开始的，并且是互相重叠的。这种组织模式下，操作系统代码（运行在管理态）和应用代码（运行在用户态）运行的时候就彷佛不存在段似的。操作系统和应用程序的代码和数据之间的保护由处理器的页级保护机制来提供。

4.11.3.页类型

页级保护机制识别两种页类型：

- 只读访问（R/W 标志为 0)。
- 可读可写访问（R/W 标志为 1)。

当处理器处于管理态并且 CR0 寄存器中的 WP 标志为 0（重启初始化以后的状态)，所有的页都是可读与可写的（写保护被忽略)。当处理器处于用户态，它只能写具有读写访问权限的用户态的页。可读写或者是只读的用户态的页都是可读的；管理态的页对用户态来说是既不可读也不可写。任何违反这些保护规则的企图都会产生一个页故

障异常。

P6 系列、Pentium 和 Intel 486 处理器可以禁止用户态的页被管理态访问。设置 CR0 寄存器中的 WP 标志为 1，可以开启管理态对用户态的写保护页的敏感性。无论 WP 如何设置，管理态的只读页在任何特权级下都是不可写的。这种管理态写保护特征对某些操作系统实现写时复制很有用，比如 UNIX 操作系统，创建任务（或者叫做 forking 或者 spawning）就是如此。当创建一个新任务时，新任务可以将父进程的整个地址空间拷贝过来。这使得子任务完全拥有和父进程一样的段和页。写时复制的另一种策略就是将子进程的段和页映射到父进程的段和页上，这样可以节省内存空间和时间。仅当其中一个任务要往页中写入时才需要为它创建一个这个页的属于它自己的复制。通过使用 WP 标志并把这些共享页标为只读，管理态可以检测到对用户态页写操作的企图，因此能够在这个时候及时复制页。

4.11.4. 联合使用两级页表的保护

对于任何一个页，它的页目录表项（第一层次的页表）的保护属性可能与页表项（第二层次页表）的保护属性不同。处理器会对一个页的页目录表项和页表项都进行保护检验。表 4-2 显示了当 WP 标志被清零后，所有保护属性的组合所能够提供的保护。

4.11.5. 取代页保护

无论当前处理器处于什么 CPL，以下类型的内存访问都被当作是特权级 0 的访问来进行检验的：

- 访问 GDT、LDT 或 IDT 中的描述符。
- 在对特权级之间的一个调用，或者对异常或中断处理程序的一个调用期间（此时是要改变特权级的），访问内部特权级栈。

4.12. 联合使用页保护和段保护

当开启分页时，处理器首先评价段保护，然后再评价页保护。如果处理器在段级或者页级检测到保护违例，就不能访问内存并且产生一个异常。如果分段产生异常，则不会产生分页异常。

页级保护不能用来取代段级保护。比如，一个代码段被定义为不可写的情况。如果该代码段被分页，并且设置 R/W 标志让这些页可读写，但是这并不能使这些页可写。对这些页进行写操作的企图将会在段级保护检测时被阻止。

页级保护可以用来强化段级保护。比如，当一个大的可读写数据段被分页了，页保护机制可以用来对某些个别的页进行写保护。

表 4-2 联合使用页目录表保护和页表保护

页目录表项		页表项		联合使用	
特权	访问类别	特权	访问类别	特权	访问类别
用户态	只读	用户态	只读	用户态	只读
用户态	只读	用户态	可读可写	用户态	只读
用户态	可读可写	用户态	只读	用户态	只读
用户态	可读可写	用户态	可读可写	用户态	可读可写
用户态	只读	管理态	只读	管理态	只读
用户态	只读	管理态	可读可写	管理态	可读可写*
用户态	可读可写	管理态	只读	管理态	可读可写*
用户态	可读可写	管理态	可读可写	管理态	可读可写
管理态	只读	用户态	只读	管理态	只读
管理态	只读	用户态	可读可写	管理态	可读可写*
管理态	可读可写	用户态	只读	管理态	可读可写*
管理态	可读可写	用户态	可读可写	管理态	可读可写
管理态	只读	管理态	只读	管理态	只读
管理态	只读	管理态	可读可写	管理态	可读可写*
管理态	可读可写	管理态	只读	管理态	可读可写*
管理态	可读可写	管理态	可读可写	管理态	可读可写

注意：

*如果 CR0 寄存器的 WP 标准被置位，访问类型由页目录表项和页表项的 R/W 标准来确定。

第 5 章 中断和异常处理

本章描述保护模式下处理器的中断和异常处理机制。这里提到的绝大多数内容同样适用于实地址模式和虚拟 8086 模式下的中断和异常处理机制。实地址模式和虚拟 8086 模式下中断和异常处理机制的区别的描述，参看第 15 章“调试和性能监测”。

5.1. 中断和异常概述

中断和**异常**是一些提示性事件，这些事件表明系统、处理器或者当前执行程序或任务中存在着某种状况，需要处理器注意。典型情况下，它们会导致当前运行程序或任务的执行强制性地转移到一个称作**中断处理程序**或**异常处理程序**的特殊软件例程或者任务中。处理器响应中断或者异常所采取的行动称为**服务**或者**处理**中断和异常。

典型情况下，中断是在程序执行期间随机发生的，是对硬件信号的响应。系统硬件使用中断去处理处理器的外部事件，比如服务外设的请求。使用 INT n 指令，软件也可以产生中断。

异常是在处理器执行指令的过程中发现错误状况而产生的，比如除数为零。处理器可以检测出多种不同的错误状况，包括保护违例、页故障、内部机器故障。Pentium 4、Intel Xeon、P6 系列和 Pentium 等处理器的**机器检测架构**还允许当检测到内部硬件错误和总线错误时产生机器检测异常。

IA-32 架构的中断和异常处理机制允许中断和异常的处理对于应用程序和操作系统或管理程序来说是透明的。当收到中断信号或检测到异常时，处理器便自动挂起当前正在运行的进程或任务，转而去执行中断或异常处理程序。中断或异常处理程序执行完之后，处理器继续执行被中断的进程或任务。被中断的进程或任务的继续执行不会损失程序的连续性，除非处理器无法从异常中恢复过来，或者中断使当前运行程序终止。

本章描述保护模式下处理器的中断和异常处理机制。本章的最后将详细描述异常及其产生的条件。实地址模式和虚拟 8086 模式下的中断和异常处理机制的描述，参看第 16 章“8086 仿真”。

5.2. 异常和中断向量

为了便于处理中断和异常，IA-32 架构定义的每一个异常和需要处理器特殊处理的每一个中断状况都分配了一个唯一的识别码，称作向量。处理器用分配给每个异常或者中断的向量作为访问中断描述符表（IDT）的索引，以确定异常或者中断处理程序的入口点所在的表项（参看 5.10.“中断描述符表（IDT）”）。

向量号的允许范围是 0 到 255。0 到 31 被 IA-32 架构保留给架构定义的异常和中断。但是并不是所有这 32 个向量目前都有定义好的功能，未用的向量留作未来使用。**不要使用这些保留的向量。**

32 到 255 之间的向量号指派给用户定义的中断使用，不在 IA-32 架构的保留之列。这些中断一般被分配给外部 I/O 设备，供它们开启那些设备，以便通过某个外部硬件中断机制（5.3.“中断源”）发送中断给处理器。

表 5-1 列出了分配给架构上定义的异常和 NMI 中断的中断向量。对每一个中断，该表给出了异常类型（参看 5.5.“异常分类”）并指出错误码是否存放在异常的栈中。还给出了异常和 NMI 中断的源。

表 5-1 保护模式异常和中断

向量号	助记符	描　述	类型	错误码	源
0	#DE	除法错	故障	没有	DIV 和 IDIV 指令
1	#DB	保留	故障/陷阱	没有	只由 Intel 使用
2	-	NMI 中断	中断	没有	不可屏蔽外部中断
3	#BP	断点	陷阱	没有	INT 3 指令
4	#OF	溢出	陷阱	没有	INTO 指令
5	#BR	BOUND 范围越界	故障	没有	BOUND 指令
6	#UD	非法操作码（未定义操作码）	故障	没有	UD2 指令或者保留的操作码 [1]
7	#NM	设备不可用（无数学协处理器）	故障	没有	浮点或者 WAIT/FWAIT 指令
8	#DF	双故障	终止	有（0）	任何一个产生异常、NMI 或 INTR 的指令
9		协处理器段超出（保留）	故障	没有	符点指令 [2]
10	#TS	非法 TSS	故障	有	任务切换或者 TSS 访问
11	#NP	段不存在	故障	有	加载段寄存器或者访问系统段
12	#SS	栈段故障	故障	有	栈操作和 SS 寄存器加载

13	#GP	一般保护	故障	有	任何内存引用和其它保护检验
14	#PF	页故障	故障	有	任何内存引用
15	-	Intel 保留，未使用	故障	没有	
16	#MF	x87 FPU 符点错误（数学故障）	故障	没有	x87 FPU 符点或者 WAIT/FWAIT 指令
17	#AC	对齐检验	故障	有（0）	任何内存中的数据引用 [3]
18	#MC	机器检验	终止	没有	错误码（如果有的话）和源是模型相关的
19	#XF	SIMD 符点异常	故障	没有	SSE/SSE2/SSE3 符点指令 [5]
20-31	-	Intel 保留，未使用			
32-255	-	用户定义（未保留）中断	中断		外部中断或者 INT n 指令

备注：

1. UD2 指令是在 Pentium Pro 处理器引入的。
2. Intel 386 之后的 IA-32 处理器不再产生这个异常。
3. 这个异常是在 Intel 486 处理器引入的。
4. 这个异常是在 Pentium 处理器引入的，并在 P6 系列处理器中得到增强。
5. 这个异常是在 Pentium III 处理器引入的。

5.3. 中断源

处理器接收到的中断有两个来源：

外部（硬件产生的）中断。

软件产生的中断。

5.3.1. 外部中断

外部中断是通过处理器的引脚或者本地 APIC 接收的。Pentium 4、Intel Xeon、P6 系列和 Pentium 等处理器上的主中断引脚是 LINT[1：0]这两个引脚，它们连接到本地 APIC（参看 7.5.“高级可编程中断控制器（APIC）”）。当 APIC 打开时，可通过 APIC 的局部向量表（LVT）对 LINT[1:0]引脚编程，使其和处理器的任意异常和中断向量关联。

当本地 APIC 被关闭时，这两个引脚被分别配置成 INTR 和 NMI 引脚。当激活 INTR 引脚传递一个信号给处理器时，则表明有一个外部中断发生了，处理器从系统总线读取由诸如 8259A 这样的外部中断控制器（，参考 5.2.“异常和中断向量”）发来的中断

向量号。激活 NMI 引脚则发送的是一个不可屏蔽中断（NMI）的信号，其向量号为 2。

通常，处理器的本地 APIC 通常是连接到系统的 I/O APIC 上。因而，I/O APIC 引脚接收到的外部中断是通过系统总线（Pentium 4 和 Intel Xeon 等处理器）或者 APIC 串行总线（P6 系列和 Pentium 等处理器）定向到本地 APIC 上。I/O APIC 确定中断向量号并将其送往本地 APIC。当一个系统拥有多个处理器时，处理器之间也可以通过系统总线（Pentium 4 和 Intel Xeon 等处理器）或者 APIC 串行总线（P6 系列和 Pentium 等处理器）相互发送中断。

Intel 486 和早期的 Pentium 处理器不具有芯片内建的本地 APIC，因而也没有 LINT[1：0]引脚。这些处理器却有专用的 NMI 和 INTR 引脚。对于这些处理器，外部中断由系统板上的中断控制器（8259A）产生，中断由 INTR 引脚传递信号给处理器。

注意，还有其它几个处理器引脚也可以产生处理器中断，但本章描述的中断和异常处理机制并不适用于这些中断的处理。这些引脚包括：RESET#、FLUSH#、STPCLK#、SMI#、R/S#和 INIT#。哪些处理器中包括哪些这些引脚，是与具体的 IA-32 处理器的实现相关的。这些引脚的功能在各自处理器的数据手册中有描述。第 13 章“系统管理”也有对 SMI#引脚的描述。

5.3.2. 可屏蔽硬件中断

凡是通过 INTR 引脚或本地 APIC 传递到处理器的外部中断都被称作可屏蔽硬件中断。通过 INTR 引脚传递的可屏蔽硬件中断可使用 IA-32 架构定义的所有中断向量(0 到 255)；而通过本地 APIC 传递的只能是 16 到 255 号向量。

使用 EFLAGS 寄存器的 IF 标志可以一下子屏蔽全部可屏蔽硬件中断(参看 5.8.1.“屏蔽可屏蔽硬件中断”)。注意当 0 到 15 号中断通过本地 APIC 传递时，APIC 会指出收到一个非法向量。

5.3.3. 软件产生的中断

将中断向量号作为 INT 指令的操作数即可通过 INT 指令在软件中产生中断。比如，指令 INT 35 即可强制显式地调用第 35 号中断处理例程。

0 到 255 中的任何一个都可用作 INT n 指令的参数。但是，如果使用处理器预先定义的 NMI 向量，则处理器的响应将不同于正常情况下 NMI 中断发生时的响应。如果把

2(NMI 向量)用在 INT n 指令中，则 NMI 处理例程被调用，但是处理器的 NMI 处理硬件并未被激活。

注意

EFLAGS 寄存器中的 IF 标志不能屏蔽软件中用 INT n 指令产生的中断。

5.4. 异常源

处理器接收的异常信号有三个来源：

处理器探测到的程序错误异常。

软件产生的异常。

机器检测异常。

5.4.1. 程序错误异常

在应用程序、操作系统或管理程序的执行过程中，当探测到程序错误时，处理器产生一个或多个异常。IA-32 架构为处理器可探测到的每个异常定义了一个向量号。这些异常又进一步被划分为**故障、陷阱**和**终止**（参看 5.5. “异常分类”）。

5.4.2. 软件产生的异常

INTO、INT 3 和 BOUND 指令允许在软件中产生异常。这些指令允许在指令流中的指定点检测指定的异常条件。例如，INT 3 产生一个断点异常。

INT n 指令可以在软件中用来模拟某个异常，但有一点限制。若指令中的 n 操作数包含的是一个针对 IA-32 架构异常的向量，则处理器会产生一个指向那个向量的中断，并调用与那个向量相关的异常处理例程。但是，因为这其实是一个中断，所以，处理器并不将错误码压入栈中，尽管正常情况下硬件产生这个向量的异常时会有一个错误码产生。对于那些产生错误码的异常来说，处理例程在处理异常时都会试图从栈中弹出一个错误码的。如果用 INT n 指令模仿一个异常的产生，则处理例程会弹出并丢弃 EIP（代替丢失的错误码），并让程序返回到错误的地方。

5.4.3. 机器检测异常

P6 系列和 Pentium 等处理器同时提供了内部和外部的机器检测机制，用来检查内部芯片硬件和总线事务的操作。这些机制构成了扩展的异常机制（具体情况依赖于不同的处理器）。当探测到一个机器检测错误时，处理器发出机器检测异常（18 号向量）信号，并返回一个错误码。有关机器检测机制的详细描述，参考本章后面的“18 号中断——机器检测异常（#MC）”和第 14 章“机器检测架构”。

5.5. 异常分类

根据报告的方式和引起异常的指令能否在不失程序或者任务连续性的情况下重新执行，异常被分为**故障**、**陷阱**和**终止**三种。

故障　故障是一种通常能够被修正的异常，而且，一旦修正，程序能够在不失连续性的情况下重新启动。当有故障报告时，处理器将机器状态恢复到发生故障指令之前的状态。故障处理例程的返回地址（CS 寄存器和 EIP 寄存器的保存值）指向产生故障的指令，而不是产生故障指令之后的那条指令。

陷阱　陷阱是一种在引起陷阱的指令执行之后马上报告的一种异常。陷阱允许程序或者任务的执行在不失连续性的情况下继续进行。陷阱处理例程的返回地址指向引起陷阱指令的下一条指令。

终止　终止是一种并不总是报告引起异常的指令的确切位置的异常，也不允许引起异常的进程或任务重新执行。终止被用来报告严重错误，比如硬件错误、不一致或系统表中的非法值。

注意

只有少数几个报告为故障的异常不能重新开始，并导致处理器状态丢失。一个例子是，执行 POPAD 指令时如果栈的指针越过了栈段的尾部，则将报告一个这样的故障。这里，异常处理例程将会看到指令指针（CS：EIP）恢复原样，就好象 POPAD 从未执行。但内部处理器状态（尤其是通用寄存器）却被改变了。这种陷入绝境的情况被视为编程错误，引起这种异常的应用程序很可能会被操作系统终止。

5.6.程序或任务重新开始

为了在中断或异常处理之后重新开始程序或任务，要确保除“终止”之外的所有异常均严格地在指令边界（前一条指令结束而下一条指令未开始的地方——译者）处报告，所有的中断也是如此。

对于故障类的异常，处理器是在产生指向发生故障指令的异常的同时，保存返回指令指针的。所以当程序或者任务在故障处理之后重新开始时，重新开始（重新执行）的是产生故障的指令。重新开始产生故障的指令通常用来处理当访问操作数受挫时的异常情况。最常见的例子是页故障异常（#PF），它发生在进程或任务访问不在内存中的页时。一旦有页故障异常发生，页故障异常处理例程就会把相应的页加载到内存，并重新执行程序或者任务中产生故障的指令。为确保这个指令的重新开始的处理对当前执行程序或任务来说是透明的，处理器保存必要的寄存器和栈指针，以便被中断的进程或任务恢复到产生故障指令执行之前的状态。

对陷阱类异常来说，返回指令指针指向产生陷阱指令之后的第一条指令。如果在转移指令执行过程中检测到陷阱，则返回指令指针要反映这个转移。例如，在执行 JMP 指令时探测到有陷阱异常，返回指令指针则指向 JMP 指令的目的地址，而不是 JMP 指令之后的第一条指令。所有的陷阱异常允许进程或任务的继续执行不失连续性。例如，溢出异常就属于陷阱。当这种异常发生时，返回指令指针指向 INTO 指令（测试 EFLAGS 寄存器的 OF 标志（溢出））后的那条指令。该异常的陷阱处理例程处理溢出情况。从陷阱处理例程返回时，进程或任务从 INTO 指令的后一条指令处开始执行。

终止类异常不保证进程或任务重新开始的可靠性。通常终止处理例程的作用是：在终止异常发生时收集有关处理器状态的各种诊断信息，并尽可能正常地关闭应用程序和系统。

中断必须绝对支持不失连续性的条件下重新开始被中断的进程和任务。保存的返回指令指针指向发生中断的下一条指令。如果将要执行的指令带有重复前缀，则中断发生在相关寄存器当前叠代结束下次叠代开始执行之前。

P6 系列处理器的推测性执行指令的能力不会影响中断的发生。发生在指令边界的中断是在指令执行的退役阶段（retirement phase）定位的。因此，它们通常发生在“按序”指令流中。有关 P6 系列处理器的微架构和对无次序指令执行的支持，参看《第 1 卷：基础架构》的第 2 章“Intel 架构简介”。

注意，Pentium 处理器和早期的 IA-32 处理器也实行大量的预取指令和预先指令解码。但是，这个时候是不会发出中断和异常信号的，直到指令实际“按序”执行才会。就给定的代码而言，异常信号的发出都是当指令在任何一个 IA-32 系列处理器上执行时才会出现（除非这是一个新定义的异常或者操作码）。

5.7.不可屏蔽中断（NMI）

在两种情况下将产生不可屏蔽中断（NMI）：

外部硬件激活 NMI 引脚。

处理器从系统总线（Pentium 4 和 Intel Xeon 处理器）收到消息或者从 APIC 串行总线（P6 系列和 Pentium 处理器）收到送达模式的 NMI 消息。

当处理器从这两种中的任一种收到 NMI 时，便立即作出响应，调用由 2 号中断向量指向的处理例程。处理器还会调整某些硬件以保证在当前 NMI 处理例程完成前，不再接收任何中断信号，包括 NMI 中断（参看 5.7.1.“处理多个 NMI”）。

当从上述两个来源中接收 NMI 时，它是不能被 EFLAGS 寄存器的 IF 标志屏蔽的。

发送一个可屏蔽硬件中断（通过 INTR 引脚）到 2 号向量来调用 NMI 处理例程是可能的。但是，这种中断不是真正的 NMI 中断。真正的激活处理器 NMI 处理硬件的中断，只能由上面提到的两种机制之一产生。

5.7.1.处理多个 NMI

在 NMI 处理例程执行的过程中，处理器会禁止其它对 NMI 处理程序的调用，直到执行了下一个 IRET 指令。对后续 NMI 的堵塞会阻止对 NMI 处理程序的访问的堆叠。建议使用中断门来调用 NMI 中断处理例程，以禁止可屏蔽硬件中断（参看 5.8.1.“屏蔽可屏蔽硬件中断”）。如果 NMI 处理程序是 IOPL 小于 3 的虚拟 8086 任务，从中发出的 IRET 指令会产生一般保护异常（参看 16.2.7.“敏感指令”）。这种情况下，在一般保护异常处理程序调用之前不屏蔽 NMI。

5.8.打开和关闭中断

根据处理器和 EFLAGS 寄存器的 IF 标志和 RF 标志的状态，处理器禁止某些中断的

产生，详见下面的描述。

5.8.1.屏蔽可屏蔽硬件中断

IF 标志可以关闭对处理器的 INTR 引脚或者本地 APIC 收到的（参看 5.3.2.“可屏蔽硬件中断”）可屏蔽硬件中断的服务。当 IF 标志清除时，处理器禁止中断传到 INTR 引脚或者通过本地 APIC 产生内部中断请求；当 IF 标志置位时，中断传送到 INTR 引脚或者通过本地 APIC 引脚处理成正常外部中断。

IF 标志不影响传送到 NMI 引脚的不可屏蔽中断（NMI）和通过本地 APIC 传送的送达模式的 NMI 消息，也不影响处理器产生异常。和 EFLAGS 寄存器的其它标志一道，处理器清除 IF 标志以响应硬件复位。

可屏蔽硬件中断包括保留中断和 0 到 32 异常向量，这两种状况会引起混淆。从构架上说，当 IF 标志置位时，0 到 32 号向量之间的任何中断都可以通过 INTR 引脚传送给处理器，16 到 32 之间的任何一个都可通过本地 APIC 传送。处理器随后产生一个中断，并调用相应的向量号所指的中断或者异常处理程序。因此，就例子而言，可以通过 INTR 引脚（通过 14 号向量）调用页故障处理程序。但是，这不是真正的页故障异常。这是一个中断。由于使用 INT n 指令（参看 5.4.2.“软件产生的异常”），此时中断的产生是通过 INTR 引脚赋给一个异常向量的，处理器就不把一个错误号压入栈中，因此，异常处理程序不能保证正确运行。

IF 标志分别用 STI（开启中断允许标志）和 CLI（清除中断允许标志）指令设置或清除的。只有当 CPL 小于或等于 IOPL 时才可以执行这两个指令。如果在 CPL 大于 IOPL 的情况下执行，将会产生一个一般保护异常（#GP）。（当通过设置 CR4 寄存器的 VME 标志开启虚拟模式扩展时，IOPL 对这两个指令的影响略有变化。关于这一点参看 16.3.“虚拟 8086 模式下的中断和异常处理”。PVI 标志也会影响它的行为，参看 16.4.“保护模式虚拟中断”。）

IF 标志也受下列操作的影响：

PUSHF 指令保存所有的标志进入栈中，这里会检查并修改它们。POPF 指令用来把修改后的标志装入 EFLAGS 寄存器。

任务切换、POPF 和 IRET 指令都会装载 EFLAGS 寄存器，因而，它们可被用来修改 IF 标志。

当中断是通过中断门处理的时候，IF 标志会自动被清除，关闭可屏蔽硬件中断。（如果中断是通过陷阱门处理的，IF 标志不被清除。）

有关 CLI、STI、PUSHF、POPF 和 IRET 指令及其是否允许在 IF 标志上操作的详细描述，参看《第 2 卷：指令集参考》中的第 3 章。

5.8.2.屏蔽指令断点

EFLAGS 寄存器中的 RF（重新执行）标志控制着处理器对指令断点条件的响应（参看 2.3.“EFLAGS 寄存器中的系统标志和域”）。

当 RF 标志置位时，则阻止指令断点产生调试异常（#DB）；当 RF 标志清除时，允许指令断点产生调试异常。RF 标志的主要作用是阻止处理器在指令断点时进入调试异常循环。更多关于这个标志的信息参看 15.3.1.1.“指令断点异常条件”。

5.8.3.栈切换时屏蔽中断和异常

为了切换到不同的栈段，软件常使用成对的指令，比如：

```
MOV SS，AX
MOV ESP，StackTop
```

如果中断或者异常发生在段选择子装入 SS 寄存器之后，ESP 寄存器被装入值之前，进入栈空间的逻辑地址的这两个部分将在中断和异常处理程序期间不统一。

为了防止这种局面出现，处理器在 MOV 到 SS 指令或者 POP 到 SS 指令之后禁止中断、调试异常和单步陷阱异常，直到紧跟着的下一条指令到达的它的指令边界（也就是下一条指令执行完之后才行——译者）。

5.9.并发异常或中断的优先关系

如果在指令边界有多个异常或中断等待处理，处理器将以预定的顺序来处理它们。表 5-2 显示了各类异常和中断源的优先关系。

表 5-2 中所列的各类优先级在整个架构中是统一的，每类中的异常个数与具体的处理器实现相关，不同的处理器会有所不同。处理器首先为具有最高优先级的待处理异常或者中断服务，把执行流转移到相应处理程序的第一条指令上。较低优先级的异

常被丢弃，但较低优先级的中断继续保留待处理。当执行流返回到发生异常或者中断的程序或者任务的指令处时，被丢弃的异常会再次产生。

表 5-2 并发异常和中断的优先关系

优先级	描　　述
1（最高级）	硬件复位和机器检测 ——复位 ——机器检测
2	任务切换时的陷阱 ——TSS 中的 T 标志被设置
3	外部硬件干预 ——FLUSH ——STOPCLK ——SMI ——INIT
4	前一个指令的陷阱 ——断点 ——调试陷阱异常（TF 标志设置或者数据/I-O 断点）
5	外部中断 ——NMI 中断 ——可屏蔽硬件中断
6	代码断点故障
7	预取下条指令故障 ——代码段界限违例 ——代码页故障
8	解码下条指令故障 ——指令长度大于 15 个字节 ——非法操作码 ——协处理器不可用
9（最低级）	执行指令时的故障 ——溢出 ——边界错误 ——非法 TSS ——段不存在 ——栈故障 ——一般保护 ——数据页故障 ——对齐检验 ——x87 FPU 浮点异常 ——SIMD 浮点异常

5.10. 中断描述符表（IDT）

中断描述符表（IDT）把每一个异常或中断向量和对应的中断或者异常服务的例程或任务的门描述符之间建立了联系。同 GDT 和 LDT 一样，IDT 也是由一系列 8 字节描述符组成的（在保护模式下）。不同于 GDT 的是，IDT 中的第一项可以包含一个描述符。异常或中断向量号乘上 8(即门描述符包含的字节数)即可得到 IDT 中的描述符的索引。由于只有 256 个中断或异常向量，所以 IDT 包含的描述符不必多于 256 个。但它包含的描述符可以不足 256 个，因为只有那些确实发生的异常或中断才需要一个描述符。IDT 中所有空置的描述符必须将存在标志清零。

IDT 的基地址必须是 8 字节边界对齐的，以装满高速缓存线来最大化性能。界限值以字节为单位，其与基地址的和即为最后一个合法字节的地址。若界限值为 0，则合法字节只有一个。因为 IDT 项是 8 字节长，所以界限总是应该为 8 的整数倍减一(即 8N-1)。

IDT 可存在于线性地址空间的任意位置。如图 5-1，处理器使用 IDTR 寄存器寻址 IDT。该寄存器包含 32 位的基地址和 16 位的界限。

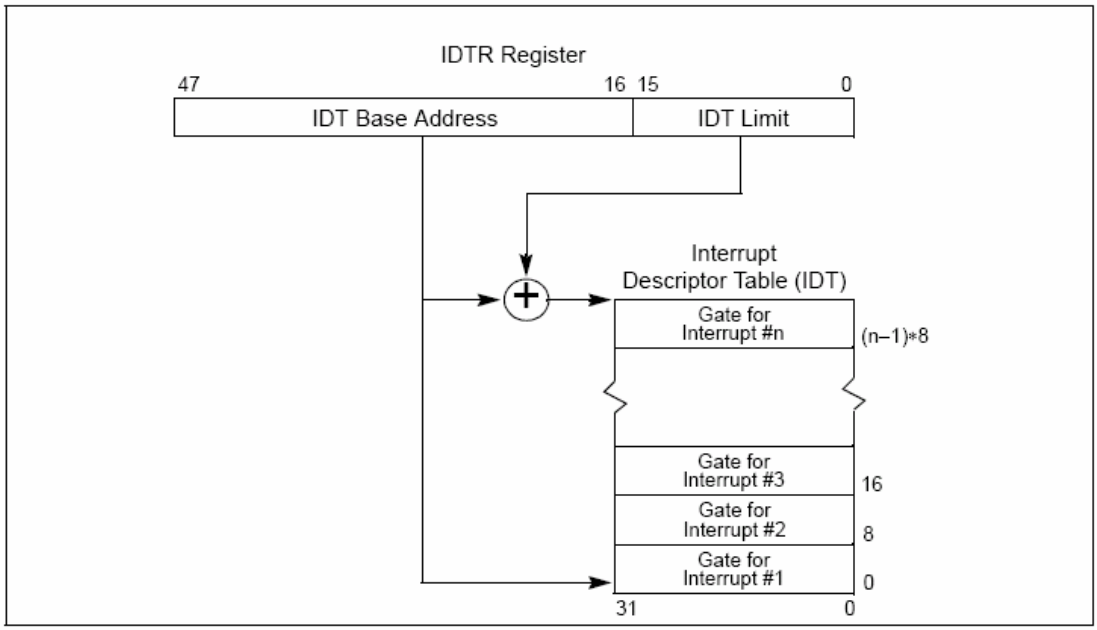

Figure 5-1. Relationship of the IDTR and IDT

LIDT（装载 IDT 寄存器）和 SIDT（保存 IDT 寄存器）指令分别用来装载和保存 IDTR 寄存器的值。LIDT 指令使用包含基地址和界限的内存操作数装载 IDTR 寄存器。该指令只有当 CPL 为 0 时才能使用。通常在操作系统的初始化代码中创建 IDT 时才被用到。操作系统也可以使用它更换 IDT。SIDT 指令将 IDTR 寄存器中的基地址和界限保存到内

存操作数中。SIDT 可在任何特权级上使用。

如果向量引用的描述符超过 IDT 的界限，将发生一般保护异常（#GP）。

5.11. IDT 描述符

IDT 中可以包含以下三种门描述符：

- 任务门描述符。
- 中断门描述符。
- 陷阱门描述符。

图 5-2 示出了任务门、中断门和陷阱门三种描述符的格式。IDT 中使用的任务门的格式同 GDT 或 LDT 中的完全一样（参看 6.2.4.“任务门描述符”）。任务门中包含异常或中断处理任务的 TSS 的段选择子。

中断门和陷阱门同调用门（参看 4.8.3.“调用门”）非常相似。它们包含一个远指针（段选择子和偏移），处理器用它来将执行流转移至异常或中断处理代码段中的处理例程。这些门在处理器处理 EFLAGS 寄存器中的 IF 标志的方式上有所不同（参看 5.12.1.2.“异常或中断处理程序对标志的使用”）。

5.12. 异常和中断处理

处理器对异常和中断调用的处理方式与用 CALL 指令调用例程和任务的处理十分相似。响应异常和中断时，处理器将异常或中断向量作为 IDT 中描述符的索引。若该索引指向一个中断门或陷阱门，那么处理器会象处理 CALL 指令引用调用门一样，引用异常或中断例程。(参考 4.8.2.“门描述符”到 4.8.6.“从被调例程返回”)。若该索引指向的是任务门，处理器则执行任务切换，切换到异常或中断处理例程，与用 CALL 指令调用一个任务门相似(参考 6.3.“任务切换”)。

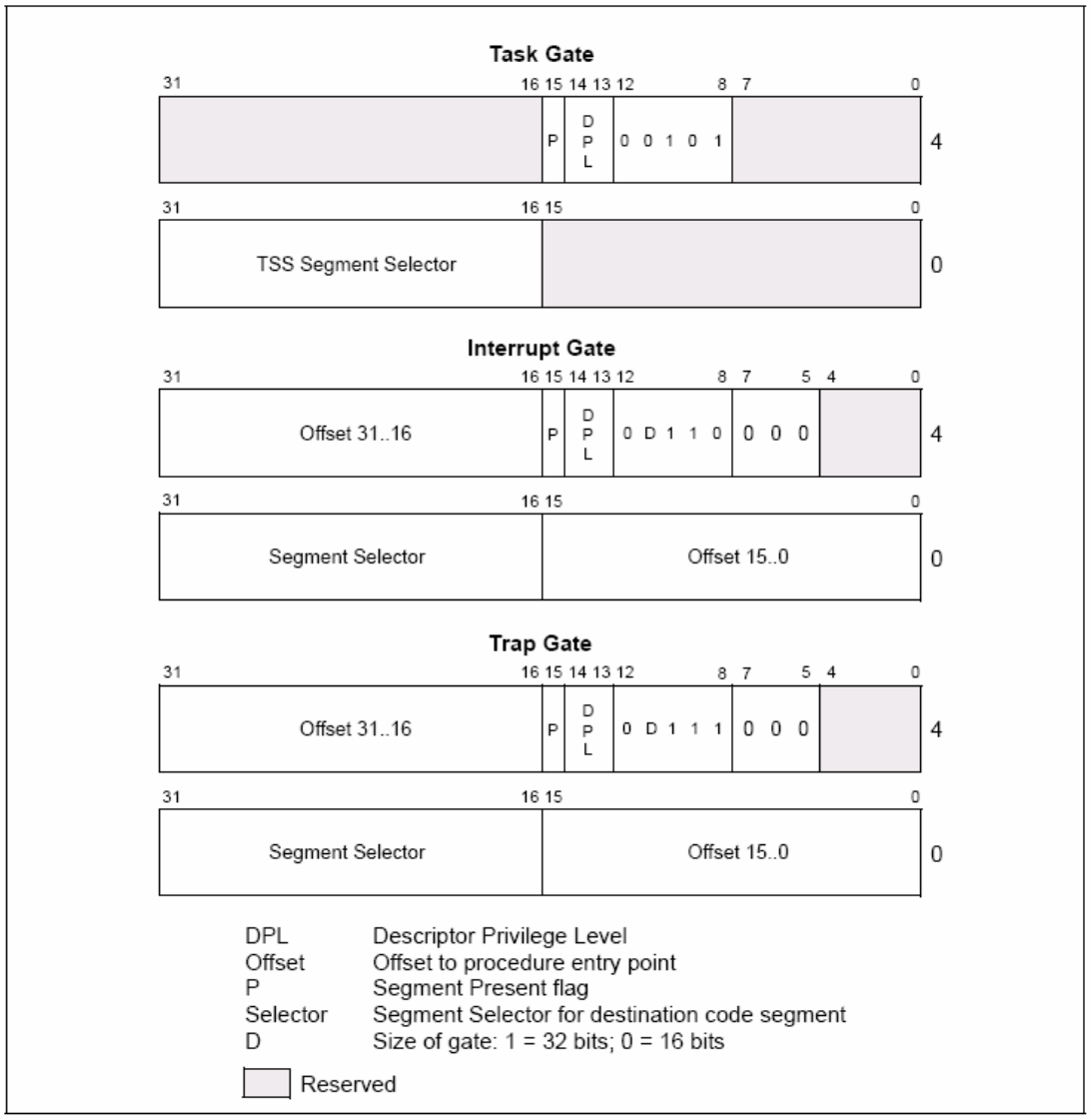

Figure 5-2. IDT Gate Descriptors

5.12.1.异常或中断处理例程

指向异常或中断处理例程的中断门或陷阱门运行在当前进程的场境中(参考图 5-3)。门的段选择子指向位于 GDT 或当前 LDT 中的可执行代码段的段描述符。门描述符中的偏移指向异常或中断处理例程的入口。

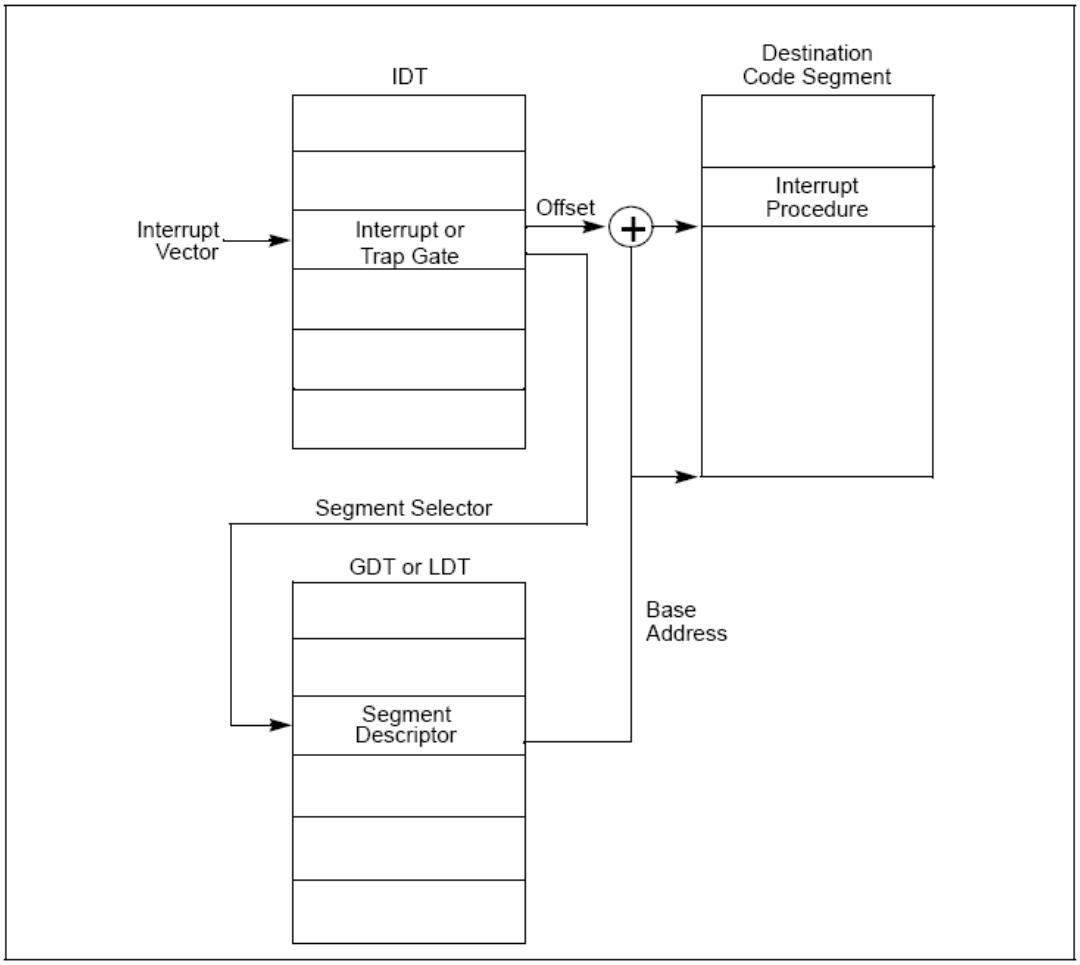

Figure 5-3. Interrupt Procedure Call

当处理器执行一个对异常或中断处理例程的调用时：

如果将要执行的处理例程特权级数值较小，就进行栈切换。当栈切换发生时：

a．处理程序使用的栈的段选择子和栈指针是从当前运行任务的 TSS 中获取的。处理器把被中断例程的栈段选择子和栈指针压入新的栈中。

b．处理器随后把 EFLAGS 寄存器、CS 寄存器、EIP 寄存器的当前值保存进新栈中（参看图 5-4）。

c．如果异常同时产生了一个错误码，则把它压入栈中，位于 EIP 之后。

如果将要执行的处理例程与被中断的例程特权级相同：

a.处理器在当前栈中保存当前 EFLAGS 寄存器、CS 寄存器和 EIP 寄存器的值(参见图 5-4)。

b．如果异常的错误码也保存在那里，则把它保存在当前栈的 EIP 值之后。

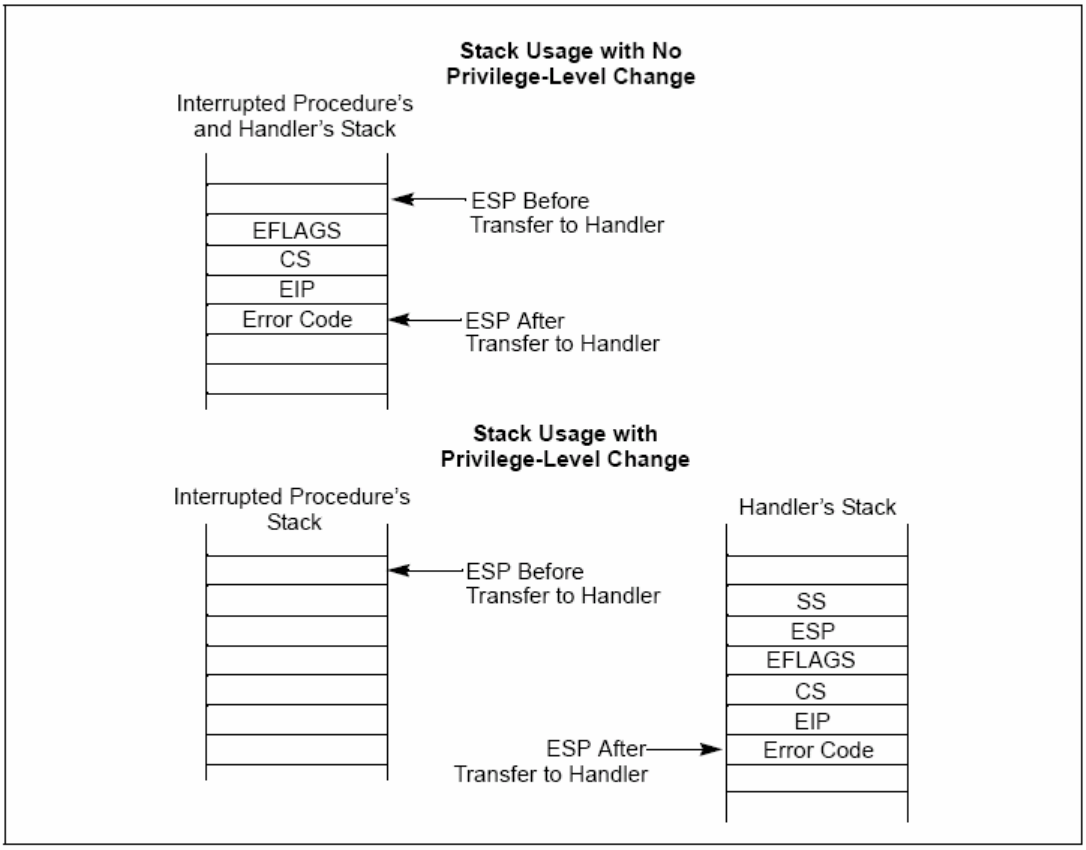

Figure 5-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

从异常或中断处理例程返回必须使用 IRET（或 IRETD）指令。IRET 指令与 RET 指令的唯一不同在于前者将恢复保存的 EFLAGS 寄存器。只有当 CPL 为 0 时，才恢复 EFLAGS 寄存器的 IOPL 域。只有 CPL 小于或等于 IOPL 时才改变 IF 标志。有关 IRET 指令的完整描述，参看《第 2 卷：指令集参考》中的第 3 章“IRET/IRETD——中断返回”。

如果在调用处理例程时发生了栈切换，则在返回时，IRET 指令还将切换回被中断例程的栈。

5.12.1.1.异常和中断处理例程的保护

异常和中断处理例程的特权级保护类似于通过调用门调用普通例程的特权级保护（参看 4.8.4.“通过调用门访问代码段”）。处理器不允许执行流转移到一个特权级低于（数值较大的特权级）当前 CPL 的异常和中断处理例程。

试图违反这个规则将导致产生一般保护异常（#GP）。异常和中断处理例程的保护机制在以下几方面有差异：

- 因为中断和异常向量没有 RPL，所以当显式地调用中断和异常处理程序时，不检验 RPL。

- 仅当中断或异常由 INT n、INT 3 或 INTO 指令产生时，处理器才检验中断或陷阱门的 DPL。此时，CPL 必须小于或等于门的 DPL。这种限制防止运行于特权级 3 的应用程序或进程使用软件中断来访问的重要的异常处理程序，如页故障处理例程，因为这些例程位于特权级更高（数值较小的特权级）的代码段中。对于由硬件产生的中断和处理器检测到的异常，处理器则忽略掉中断或陷阱门的 DPL。

异常和中断的发生通常是随机的，这些特权规则有效地为异常和中断处理例程能运行在哪些特权级上加了限制。下面提到的任一种技术都可用来避免特权级违例：

- 可以将异常或中断处理例程放到一致性代码段中。这种技术只适用于仅访问栈中数据的处理例程（例如，除法错误异常）。如果处理例程需要数据段中的数据，而此数据段必须能够被特权级 3 的程序访问，则此种情况将导致数据处于未保护之中。
- 也可以将处理例程放到特权级 0 的非一致性代码段中。这种处理例程总能够运行，而不管当前被中断进程或任务的 CPL 是多少。

5.12.1.2.异常或中断处理例程对标志位的使用

当通过中断门或陷阱门访问异常或中断处理例程时，在将 EFLAGS 寄存器的内容保存进栈后，处理器会清除 EFLAGS 寄存器的 TF 标志。（当调用异常和中断处理例程时，处理器也会在将 EFLAGS 寄存器的内容保存进栈后，清除 VM、RF 和 NT 标志。）清除 TF 标志则可以禁止指令跟踪，使中断响应不受影响。后继的 IRET 指令则使用保存在栈中的 EFLAGS 寄存器的值来恢复 TF（和 VM、RF 及 NT）标志。

中断门和陷阱门的唯一区别在于处理器处理 EFLAGS 寄存器的 IF 标志的方式。当通过中断门访问异常或中断处理例程时，处理器清除 IF 标志，以阻止另外的中断干扰当前的中断处理例程。后继的 IRET 指令用存储在栈中的 EFLAGS 的内容恢复 IF 的值。而通过陷阱门调用处理例程时，IF 标志不受影响。

5.12.2.中断任务

当通过 IDT 中的任务门访问异常或中断处理例程时，会发生任务切换。用另一个任务来处理异常或中断有下面几点好处：

- 被中断进程或任务的场境被自动保存起来。
- 处理异常或中断时，一个新 TSS 允许处理程序使用新的特权级 0 的栈。若异常或中断发生在当前特权级 0 的栈被破坏的时候，通过任务门访问处理程序由于使用了新的特权级 0 的栈而可以防止系统崩溃。
- 通过给处理程序分配一个单独的地址空间来进一步将它和其它任务隔离开。这由给它分配一个单独的 LDT 来实现的。

用独立的任务来处理中断也有不利的一面，在任务切换时，要保存大量的机器状态，这比使用中断门要慢，最终导致中断延迟。

位于 IDT 中的任务门用于引用 GDT 中的 TSS 描述符（参考图 5-5）。切换到中断或异常处理程序与普通的任务切换完全一样（参考 6.3.“任务切换”）。向后对被中断任务的链接保存在处理程序 TSS 的前一个任务链接域中。如果异常伴有错误码，则这个错误码也被拷贝到新任务的栈上。

当异常或中断处理任务被用在操作系统中时，它们实际上是两种调度任务的机制：软件调度程序（操作系统的一部分）和硬件调度程序（处理器中断机制的一部分）。当开启中断时，软件调度程序必须提供将被调度的中断任务。

注意

因为 IA-32 架构的任务是不可再入的，在中断处理任务完成处理中断的过程中和执行 IRET 指令期间，必须关闭中断。防止在中断任务的 TSS 仍然标记为忙的时候有另外的中断发生，而这会导致产生一般保护异常（#GP）。

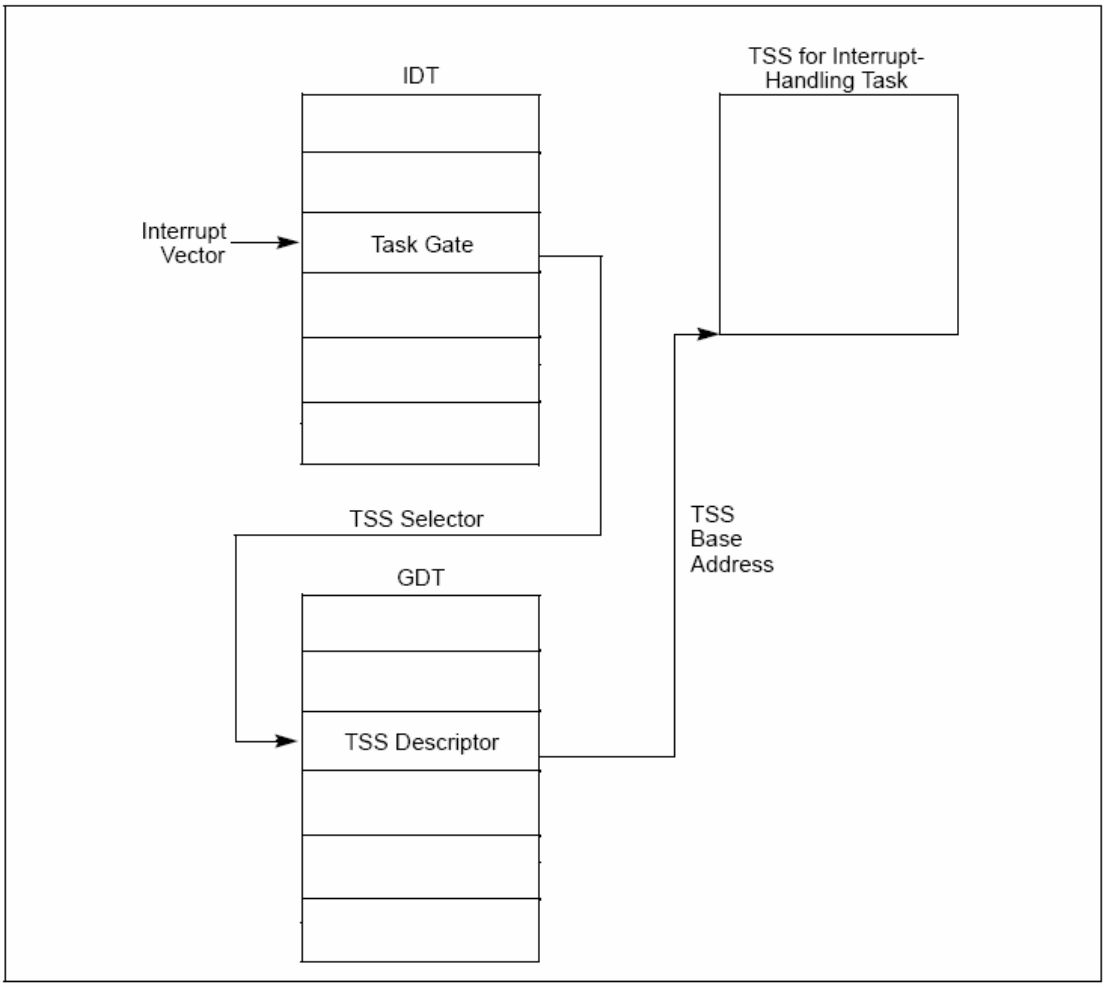

Figure 5-5. Interrupt Task Switch

5.13.错误码

当异常状态与特定的段相关时，处理器就把错误码压入异常处理程序（不管它是例程还是任务）的栈中。错误码的格式如图 5-6 所示。错误码类似一个段选择子，但是，没有 IT 标志和 RPL 域，取而代之的是 3 个标志：

EXT　　**外部事件（第 0 位）。**当置位时，则标明是程序之外的一个事件比如硬件中断等引起的异常。

IDT　　**描述符位置（第 1 位）。**当置位时，则表明错误码的索引部分指向 IDT 中的一个门描述符；当清除时，则表明指向的门描述符在 GDT 中或者当前 LDT 中。

TI　　GDT/LDT（第 2 位）。仅当 IDT 标志清除时使用。当置位时，则表明错误码的索引部分指向 LDT 中的一个段或者门描述符；当清除时，则表明指

向的当前 GDT 中的一个描述符。

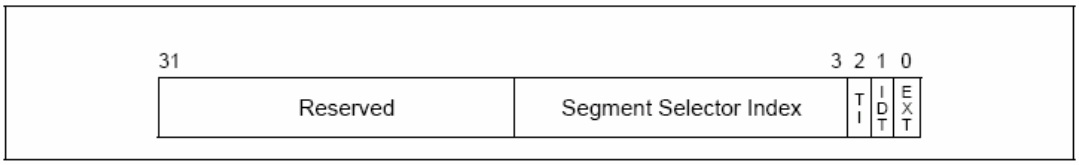

Figure 5-6. Error Code

段选择子域提供对 IDT、GDT 和当前 LDT 的索引，以确定错误码将要引用的段或者门选择子。在某些情况下，错误码是空的（也就是低半截的那个字的所有位都是 0）。空的错误码表明错误不是由指定段中的一个参考性因素引起的或者在操作中引用了一个空的段描述符。

这个错误码的格式与页故障异常（#PF）是不同的，参看本章的“14 号中断——页故障异常（#PF)”。

错误码是作为一个双字或者字（取决于默认中断、陷阱和任务门的大小）压入栈中的。为保持以双字对齐压栈，错误码的上半截保留。注意，当执行 IRET 指令从一个异常处理程序返回时，错误码是不被弹出来的。因此，处理程序必须在返回之前去除错误码。

错误码并不被压入外部产生的（INTR 或者 LINT[0:1]）或者 INT n 指令产生的异常的栈中，尽管这些异常正常情况下会产生错误码。

5.14.异常和中断参考

下面的内容描述异常和中断的产生条件。按照向量号的顺序排列。这里提到的内容包括：

异常类型 指出异常的类型是故障、陷阱还是终止。有些异常既可以是故障也可以是陷阱，这取决于何时检测到错误状态。（这部分对中断不适用。）

描述 给出对异常或中断目的的一般性描述。也描述处理器如何处理异常或中断。

异常错误码 指示是否为异常保存错误码。如果是，则描述错误码的内容。（这部分对中断不适用。）

保存的指令指针 描述保存的（返回）指令指针指向的是哪条指令。也指示该指

针是否可被用于重新开始产生故障的指令。

进程状态变化 描述异常或中断对当前执行进程或任务产生的影响，和不失连续性的情况下重新开始进程和任务的可能性。

0 号中断——除法错异常（#DE）

异常类型：故障。

描述：指出 DIV 或者 IDIV 指令的除数操作数是 0，或者结果无法用目的操作数指定的位数表示出来。

异常错误码：无。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向产生异常的指令。

进程状态变化：除法错不改变进程状态，因为异常产生在发生故障指令被执行之前。

1 号中断——调试异常（#DB）

异常类型：陷阱或者故障。通过检查 DR6 寄存器和其它调试寄存器的内容，异常处理程序可以区分是陷阱还是异常。

描述：指出几种调试异常状况中一种或者多种被测出。是故障还是陷阱取决于具体发生条件（参见表 5-3）。关于调试异常的详细描述参看第 15 章“调试和性能监视”。

表 5-3 调试异常条件和相应的异常类别

异常条件	异常类别
指令获取断点	故障
数据读或写断点	陷阱
I/O 读或者写断点	陷阱
一般检测条件（与接入电路仿真关联）	故障
单步	陷阱
任务切换	陷阱

异常错误码：无。异常处理程序会检查调试寄存器来确定哪种条件引起的异常。

保存的指令指针：故障——保存的 CS 寄存器和 EIP 寄存器的内容指向产生异常的指令。陷阱——保存的 CS 寄存器和 EIP 寄存器的内容指向产生陷阱指令的后续指令。

进程状态变化：故障——调试异常不改变进程状态，因为异常产生在发生故障指

令被执行之前。从调试异常处理程序返回之后进程可以重新开始正常的执行。陷阱——调试异常改变进程状态，因为执行的指令或者任务切换允许在异常产生之前完成。但是，进程的新状态不会被破坏，进程的执行也可以可靠地进行。

2 号中断——NMI 中断

异常类型：不用。

描述：不可屏蔽中断(NMI)是通过向处理器的 NMI 引脚发送信号或者通过 I/O APIC 向本地 APIC 设置的 NMI 请求在外部产生的。这个中断引起对 NMI 中断处理程序的调用。

异常错误码：不用。

保存的指令指针：处理器通常在指令边界获取 NMI 中断。保存的 CS 寄存器和 EIP 寄存器的内容指向中断发生点的下一条指令。更多关于处理器何时获取 NMI 中断的信息参看 5.5.“异常分类”。

进程状态变化：收到 NMI 中断的指令会在 NMI 产生之前完成执行。因而，从中断处理程序返回后的进程或者任务会在不失连续性的情况下重新开始，因为中断处理程序会在处理中断之前保存处理器的状态并在返回前恢复处理器的状态。

3 号中断——断点异常（#BP）

异常类型：陷阱。

描述：指出断点指令（INT 3）被执行了，引起了一个断点陷阱产生。典型情况下，是一个调试程序把一个指令的第一个操作码字节替换成了 INT 3 指令的操作码。(INT 3 指令的长度是一个字节，所以很容易把 RAM 中的代码段中的一个操作码替换成断点操作码。）操作系统或者调试工具可以使用一个映射到代码段所在的同样物理地址空间的数据段，在希望调用调试程序的地方放置 INT 3 指令。

在 P6 系列、Pentium、Intel 486 和 Intel 386 等处理器中，使用调试寄存器去设置断点变得更加容易。(有关断点异常的信息，参看 15.3.2.“断点异常（#BP）——中断向量 3。）如果需要的断点数量超过断点寄存器的个数，可以使用 INT 3 指令。

断点异常（#BP）也可以通过执行操作数为 3 的 INT n 指令来产生。这个指令（INT 3）的操作与 INT 3 指令稍微有所不同（参看《第 2 卷：指令集参考》第 3 章中的“INTn/INTO/INT3——对中断例程的调用”)。

异常错误码：无。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向 INT 3 指令之后的指令。

进程状态变化：尽管 EIP 指向断点指令之后的指令，但是，进程的状态实质上没有改变，因为 INT 3 指令不影响任何寄存器和内存区域。调试程序可以通过把引起断点的 INT 3 指令替换回原操作码，并把 EIP 寄存器的值减一，来重新执行挂起的程序。在从调试程序返回时，进程使用替换后的指令重新执行。

4 号中断——溢出异常（#OF）

异常类型：陷阱。

描述：指出 INTO 指令被执行了，引起了一个溢出陷阱产生。INTO 指令检验 EFLAGS 寄存器中的 OF 标志。如果 OF 标志置位，溢出陷阱就会产生。

某些算术指令（如 ADD 和 SUB）可以进行有符号和无符号的算术运算。这些指令设置 EFLAGS 寄存器的 OF 和 CF 标志，以分别指出有符号的溢出和无符号的溢出。当对有符号的操作数执行算术运算时，可以直接测试 OF 标志或者使用 INTO 指令。使用 INTO 指令的好处是，如果有溢出被测试到，可以自动调用异常处理程序去处理溢出状况。

异常错误码：无。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向 INTO 指令之后的指令。

进程状态变化：尽管 EIP 指向 INTO 指令之后的指令，但是，进程的状态实质上没有改变，因为 INTO 指令不影响任何寄存器和内存区域。在从溢出异常处理程序返回时，进程可以正常的重新执行。

5 号中断——BOUND 越界异常（#BR）

异常类型：故障。

描述：指出执行了一个 BOUND 指令，导致 BOUND 范围越界故障产生。BOUND 指令检验一个有符号的数组下标是否在数组内存地址的上界和下界之间。如果数组的下标不在数组的上下界之间，则产生一个 BOUND 越界异常。

异常错误码：无。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向产生异常的 BOUND 指令。

进程状态变化：边界检验故障不改变进程的状态，因为 BOUND 指令的操作数不会被修改。从 BOUND 范围越界异常处理程序返回后，将重新执行 BOUND 指令。

6 号中断——非法操作码异常（#UD）

异常类型：故障。

描述：指出处理器做了如下的事情之一：

- 试图执行一个非法的或者保留的操作码。
- 试图执行一个操作数类型非法的指令，比如，LES 指令的源操作数不是内存地址。
- 试图在不支持 MMX 技术或者 SSE/SSE2/SSE3 扩展的 IA-32 处理器上执行 MMX 指令或者 SSE/SSE2/SSE3 指令。CPUID 的特征标志 MMX（第 23 位）、SSE（第 25 位）、SSE2（第 26 位）、SSE3（ECX 的第 0 位）表明是否支持这些扩展。
- 试图在 CR0 寄存器的 EM 标志置位（置 1）时执行 MMX 指令或者 SSE/SSE2/SSE3 SIMD 指令（MOVNTI、PAUSE、PREFETCH*h*、SFENCE、LFENCE、MFENCE 和 CLFLUSH 等指令除外）。
- 试图在控制寄存器 CR4 的 OSFXSR 位被清除（置 0）时执行 SSE/SSE2/SSE3 指令。注意，这个不包括下述 SSE/SSE2/SSE3 指令：MOVNTI、PAUSE、PREFETCH*h*、SFENCE、LFENCE、MFENCE 和 CLFLUSH；和 64 位版本中的 PAVGB、PAVGW、PEXTRW、PINSRW、PMAXSW、PMAXUB、PMINSW、PMINUB、PMOVMSKB、PMULHUW、PSADBW、PSHUFW、PADDQ 和 PSUBQ。
- 试图在控制寄存器 CR4 的 OSXMMEXCPT 位被清除（置 0），进而导致 SIMD 浮点异常的情况下在 IA-32 处理器上执行 SSE/SSE2/SSE3 指令。
- 执行了一个 UD2 指令。注意尽管执行了一个引起非法操作码异常的 UD2 指令，但是，保存的指令指针依然指向这个 UD2 指令。
- 在一个没有加锁的指令前面测到了一个 LOCK 前缀，或者测到了带 LOCK 前缀的指令加锁了但是目的操作数却不是内存地址。
- 试图在实地址模式或者虚拟 8086 模式执行 LLDT、SLDT、LTR、STR、LSL、LAR、

VERR、VERW 或者 ARPL 等指令。

- 试图在非 SMM 模式下执行 RSM 指令。

在 Pentium 4、Intel Xeon、P6 系列等的处理器中，直到试图去移除执行非法指令的结果时才会发出这个异常。也就是说，解码和推测性执行一个非法操作码时不产生这个异常。同样，在 Pentium 处理器和早期的 IA-32 处理器中，在预取和初步解码一个非法指令期间不会产生这个异常。（有关获得中断和异常的一般性规则参看 5.5.“异常分类”。）

D6 和 F1 是 IA-32 架构保留的未定操作码。这两个操作码不会产生一个非法操作码异常。

UD2 指令肯定要产生非法操作码异常的。

异常错误码：无。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向产生这个异常的指令。

进程状态变化：非法操作码故障不改变进程的状态，因为非法指令不会被执行。

7 号中断——设备不可用异常（#NM）

异常类型：故障。

描述：指出有下列情况之一发生：

有三种情况会产生设备不可用异常：

- 在 CR0 寄存器的 EM 标志置位（置 1）时，处理器执行了 x87 FPU 符点指令。参看下一个段落有关 WAIT/FWAIT 指令特殊情况的说明。
- 在 CR0 寄存器的 MP 和 TS 标志置位时，此时不管 EM 标志的情况，处理器执行了 WAIT/FWAIT 指令。
- 在 CR0 寄存器的 TS 标志置位，EM 标志清除的情况下，处理器执行了 x87 FPU、MMX 或者 SSE/SSE2/SSE3 指令（MOVNTI、PAUSE、PREFETCH*h*、SFENCE、LFENCE、MFENCE 和 CLFLUSH 等指令除外）。

当处理器没有内置的 x87 FPU 符点单元时，EM 标志置位。此时，每当遇到 x87 FPU 符点指令，就产生一个设备不可用异常，启动异常处理程序去调用符点指令仿真例程。

TS 标志表明自从上次 x87 FPU 符点、MMX 或者 SSE/SSE2/SSE3 等指令执行之后，已经发生了一次场境切换（任务切换），但是，x87 FPU、MMX 和 MXCSR 寄存器还没有保

存。在 TS 标志置位并且 EM 标志清除的情况下，每当遇到一个 x87 FPU 符点、MMX、SSE/SSE2/SSE3 等指令时，处理器就产生一个设备不可用异常（前面列出的除外指令除外）。然后，异常处理程序就在执行指令之前保存 x87 FPU、MMX 和 MXCSR 寄存器的场境。有关 TS 标志的更多信息，参看 2.5.“控制寄存器”。

CR0 寄存器的 MP 标志用来和 TS 标志一起来确定 WAIT 或者 FWAIT 指令是否应该产生设备不可用异常。它把 TS 标志的作用扩展到 WAIT 和 FWAIT 指令上，为异常处理程序提供一次在执行 WAIT 或者 FWAIT 指令之前保存 x87 FPU 场境的机会。MP 标志主要提供给 Intel 286 和 Intel 386 DX 处理器使用的。就运行在 Pentium 4、Intel Xeon、P6 系列、Pentium、Intel 486 DX 处理器或者 Intel 486 SX 协处理器的程序而言，MP 标志应该总是置位；就运行在 Intel 486 SX 处理器的程序而言，MP 标志应该被清除。

异常错误码：无。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向产生异常的浮点指令或者 WAIT/FWAIT 指令。

进程状态变化：设备不可用故障不改变进程的状态，因为产生异常的指令不被执行。

如果 EM 标志置位，则异常处理程序就读取 EIP 寄存器所指的浮点指令并调用相应的仿真例程。

如果 MP 和 TS 标志置位或者仅 TS 标志置位，异常处理程序就保存 x87 FPU 的场境，清除 TS 标志，继续执行被中断的浮点指令或者 WAIT/FWAIT 指令。

8 号中断——双故障异常（#DF）

异常类型：终止。

描述：指出处理器在调用前一个异常处理程序的时候又检测到第二个异常。正常情况下，当处理器准备去调用一个异常处理程序的时候检测到另外一个异常时，这两个异常会被按顺序处理。但是，如果处理器不能顺序处理它们，就发出双故障异常信号。为确定什么时候把两个故障作为双故障异常发出一个信号，处理器把异常分为三类：良性异常、连带责任异常、页故障（参看表格 5-4）。

表 5-4 中断和异常类

类	向量号	描述

良性异常和中断	1	调试
	2	NMI 中断
	3	断点
	4	溢出
	5	BOUND 范围越界
	6	非法操作码
	7	设备不可用
	9	协处理器段越出
	16	浮点错误
	17	对齐检验
	18	机器检验
	19	SIMD 浮点
	全部	INT n
	全部	INTR
连带责任异常	0	除法错误
	10	非法 TSS
	11	段不存在
	12	栈故障
	13	一般保护
页故障	14	页故障

表 5-4 显示了引起双故障产生的异常类的各种组合。双故障异常属于终止类异常。程序或者任务不可能再重新开始或者继续执行了。双故障处理程序可以用来收集关于机器的诊断信息，或者在可能的情况下，关闭应用程序或者系统或者重新启动系统。

在预取指令的时候有可能遇到段或者页故障，但是，这种情况没有包括在表 5-5 中。在处理器试图转移控制到相关的故障处理程序的时候进一步产生的故障仍然会放到双故障队列中。

表 5-5 产生双故障的条件

第一异常	**第二异常**		
	良性的	**连带责任的**	**页故障**
良性的	顺次处理异常	顺次处理异常	顺次处理异常
连带责任的	顺次处理异常	产生一个双故障	顺次处理异常
页故障	顺次处理异常	产生一个双故障	产生一个双故障

如果在试图调用双故障处理程序时另外一个异常产生了，则处理器就进入宕机模式。这种模式类同于 HLT 指令执行之后的状态。在这种模式中，处理器停止执行指令，直到遇到 NMI 中断、SMI 中断、硬件复位或者收到 INIT#为止。处理器产生一个特殊的总线周期来说明它已经进入宕机模式。软件设计人员需要意识到当进入宕机模式时的硬件响应。比如，硬件可能会打开前面板上的指示灯，产生一个 NMI 中断去记录诊断

信息，调用复位初始化程序，产生一个 INIT 初始化程序，或者产生一个 SMI。如果宕机过程中有任何待处理事件，它们将在宕机处理过后的一个唤醒事件之后进行处理（例如，A20M#中断）。

如果宕机发生在处理器正在执行 NMI 中断处理程序的时候，那么，只有硬件复位可以重新启动处理器。同样，如果宕机发生在 SMM 模式中的执行过程中，必须使用一个硬件复位来重新开始处理器。

异常错误码：0。处理器总是把错误码 0 压入双故障处理程序的栈中。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容未定义。

进程状态变化：双故障异常后的进程的状态未定义。进程或者任务不能重新开始或者重新启动。双故障异常处理程序唯一可做的事是收集用于诊断的所有的场境信息，并且关闭应用程序或者宕机或者复位处理器。

如果在异常处理机器状态的一部分被损坏的时候发生双故障异常，那么，就不能调用处理程序，并且处理器必须复位。

9 号中断——协处理器段超出

异常类型：终止。（Intel 保留，未使用。最近的 IA-32 处理器不产生这个异常。）

描述：指出以 Intel 386 CPU 为基础的带有 Intel 387 数学协处理器的系统在传送 Intel 387 数学协处理器操作数的中间部分时，检测到段或者页违例。P6 系列、Pentium 和 Intel 486 处理器不产生这个异常；相反，这种情况是随着中断 13“一般保护异常”一同被检测的。

异常错误码：无。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向产生这个异常的指令。

进程状态变化：协处理器段超越异常后的处理器状态未定义。进程或者任务不可能继续执行或者重新开始。异常处理程序唯一可做的就是保存指令指针并使用 FNINIT 指令再次初始化 x87 FPU。

10 号中断——非法 TSS 异常（#TS）

异常类型：故障。

描述：指出有一个与 TSS 相关的错误。这种错误可在任务切换或者使用 TSS 的信

息执行指令期间检测到。表 5-6 列出了非法 TSS 异常产生的条件。

表 5-6 非法 TSS 条件

错误码索引	非法条件
TSS 段选择子索引	32 位 TSS 段的界限小于 67H 或者 16 位 TSS 段的界限小于 2CH
TSS 段选择子索引	在 IRET 的任务切换期间，TSS 段选择子的 TI 标志表示 LDT 段
TSS 段选择子索引	在 IRET 的任务切换期间，TSS 段选择子超过了描述符表的界限
TSS 段选择子索引	在 IRET 的任务切换期间，TSS 描述符的忙标志表示非活动段
TSS 段选择子索引	在 IRET 的任务切换期间，出现加载后向链接故障
TSS 段选择子索引	在 IRET 的任务切换期间，后向链接是空选择子
TSS 段选择子索引	在 IRET 的任务切换期间，后向链接指向一个非忙的 TSS 的描述符
TSS 段选择子索引	新的 TSS 描述符超出了 GDT 界限
TSS 段选择子索引	新的 TSS 描述符不可写
TSS 段选择子索引	保存旧的 TSS 时遇到一个故障条件
TSS 段选择子索引	在跳转或者 IRET 的任务切换时，旧的 TSS 描述符不可写
TSS 段选择子索引	在调用或者异常的任务切换时，新的 TSS 的后向链接不可写
TSS 段选择子索引	在试图锁定新的 TSS 时，发现新的 TSS 选择子是空的
TSS 段选择子索引	在试图锁定新的 TSS 时，发现新的 TSS 选择子的 TI 标志是置位的
TSS 段选择子索引	在试图锁定新的 TSS 时，发现新的 TSS 描述符不是可用描述符
LDT 段选择子索引	LDT 或者 LDT 不存在
栈段选择子索引	栈段选择子超出了描述符表的界限
栈段选择子索引	栈段选择子是空的
栈段选择子索引	栈段描述符不是一个数据段
栈段选择子索引	栈段不可写
栈段选择子索引	栈段的 DPL！=CPL
栈段选择子索引	栈段选择子的 RPL！=CPL
代码段选择子索引	代码段选择子超出了描述符表界限
代码段选择子索引	代码段选择子是空的
代码段选择子索引	代码段描述符的类型不是代码段
代码段选择子索引	非一致性代码段的 DPL！=CPL
代码段选择子索引	一致性代码段的 DPL>=CPL
数据段选择子索引	数据段选择子超过了描述符表界限
数据段选择子索引	数据段描述符类型不是一个可读代码段或者数据段
数据段选择子索引	数据段描述符的类型是非一致性代码段且 RPL>DPL
数据段选择子索引	数据段描述符的类型是非一致性代码段且 CPL>DPL
TSS 段选择子索引	TSS 段选择子对 LTR 来说是空的
TSS 段选择子索引	TSS 段选择子对 LTR 来说是 TI 置位的
TSS 段选择子索引	TSS 段描述符/上界描述符超过了 GDT 段界限
TSS 段选择子索引	TSS 段描述符不是一个可用的 TSS 类型

这个异常在原任务的场境或者新任务的场境中都可产生（参看 6.3.“任务切换”）。直到处理器完整地检验了新的 TSS 的存在，否则将在原任务的场境中产生这个异常。

一旦新 TSS 的存在得到了证实，任务切换才会被认为完成了。此后测试到的任何非法 TSS 条件都将在新任务的场境中处理。(只有当任务寄存器中加载了新 TSS 的段选择子，并且如果切换预期是对一个例程或者中断调用，则新 TSS 的前一个任务链接是指向旧的 TSS 的时候，才认为任务切换完成了。)

非法 TSS 处理程序必须是一个通过调用门调用的任务。不推荐在发生故障的 TSS 场境中处理这个异常，是因为处理器状态可能不一致。

异常错误码：包含发生违例的段的段选择子对段描述符的索引的一个错误码被压入异常处理程序的栈中。如果 EXT 标志置位，则表明异常是由于当前运行程序的外部事件引起的（比如，如果一个使用调用门的外部异常处理程序试图以任务切换的方式到一个非法的 TSS)。

保存的指令指针：如果在任务切换过程之前检测到异常条件时，则保存的 CS 寄存器和 EIP 寄存器的内容指向调用任务切换的指令。如果在任务切换过程之后检测到异常条件时，则保存的 CS 寄存器和 EIP 寄存器的内容指向新任务的第一条指令。

进程状态变化：非法 TSS 处理程序从故障恢复的能力取决于引起故障的错误条件。更多关于任务切换过程及其恢复能力参看 6.3.“任务切换”。

如果在任务切换时发生非法 TSS 异常，则它可以发生在提交新任务的这个点的前后。如果它发生在提交点之前，进程状态不会改变。如果它发生在提交点之后（此时新段选择子所指新段描述符的信息已经加载进段寄存器中)，处理器将在产生异常之前从新的 TSS 中加载所有状态信息。任务切换期间，处理器首先从新的 TSS 中把所有段选择子加载到段寄存器中，然后检验它们内容的合法性。如果发现非法 TSS 异常，其余的段选择子还会继续加载，但不是检查它们的合法性，因而对引用内存来说可能没用。非法 TSS 处理程序不应该依赖于使用这些 CS、SS、DS、ES、FS、GS 寄存器中的段选择子，以免引起另外的异常。异常处理程应该在继续执行新任务之前加载所有的段寄存器。否则的话，一般保护异常（#GP）会使得后续诊断更加困难。Intel 推荐的方法是把非法 TSS 异常处理程序作为一个任务。从非法 TSS 异常处理任务切换会被中断的任务以后将引起处理器检验寄存器，当它们从 TSS 中被加载的时候。

11 号中断——段不存在（#NP）

异常类型：故障。

描述：指出段或者门描述符的存在标志被清除了。处理器可能在下列操作中产生该异常：

- 在加载 CS、DS、ES、FS、GS 寄存器的过程中。（在加载 SS 寄存器的过程中检测到段不存在会产生栈故障异常（#SS）。）此种状况可能会出现在任务切换过程中。
- 在使用 LLDT 指令加载 LDTR 寄存器的过程中。任务切换过程中加载 LDTR 寄存器检测到 LDT 不存在时产生非法 TSS 异常（#TS）。
- 执行 LTR 指令并且 TSS 标为不存在的时候。
- 在试图使用一个标为段不存在门描述符和 TSS 的时候。但是，相反情况却是合法的。

操作系统典型情况下使用段不存在异常在短层次上实现虚拟内存。如果异常处理程序加载段并返回，则被中断的进程或者任务可以继续重新执行。

但是，门描述符中的不存在标志并不表示段真的不存在（因为门不与段相对应）。操作系统可以使用门描述符中的存在标志来触发对操作系统具有特殊作用的异常。

随后引用了不存在段的连带责任异常或者页故障将引起一个双故障异常（#DF）而不是段不存在异常（#NP）。

异常错误码：包含发生违例的段的段选择子对段描述符的索引的一个错误码被压入异常处理程序的栈中。如果 EXT 标志置位，则表明异常是由于两者之一产生的：

- 引起中断的外部事件（NMI 或者 INTR），随后引用了一个不存在的段。
- 一个良性异常，随后引用了一个不存在的段。

如果错误码执向一个 IDT 表项，则 IDT 标志置位。这发生在为中断服务的 IDT 表项引用一个不存在的段。这种事件可由 INT 指令或者硬件中断产生。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容正常情况下指向产生异常的指令。如果异常发生在为新的 TSS 中的段选择子加载段描述符时，则 CS 寄存器和 EIP 寄存器指向新任务的第一条指令。如果异常发生在访问门描述符的时候，CS 寄存器和 EIP 寄存器指向掉作用这个访问的指令（比如引用一个调用门的 CALL 指令）。

进程状态变化：如果段不存在异常的发生是由于加载寄存器（CS、DS、SS、ES、FS、GS、LDTR）的缘故，异常会造成进程状态的改变，因为寄存器没有被加载。从这种异常恢复是可能的，只要简单地把消失的段加载到内存并且在段描述符中设置存在标志。

如果段不存在异常发生在访问门描述符的过程中，异常不会造成进程状态的改变。从这种异常恢复是可能的，只要把门描述符中的存在标志设上。

如果段不存在异常发生在任务切换过程中，则它可以发生在提交新任务这个点的前和后（参看 6.3.“任务切换”）。如果发生在提交点之前，进程状态不改变。如果发生在提交点之后，则处理器将从新的 TSS 中加载所有状态信息（不再执行额外的界限、存在或类型检验）。

12 号中断——栈故障异常（#SS）

异常类型：故障。

描述：指出检测出下列与栈相关的条件之一：

- 在引用 SS 寄存器的操作期间检测出界限违例。能引起界限违例的操作包括：以栈为基础的指令如 POP、PUSH、CALL、RET、IRET、ENTER 和 LEAVE，还包括显式或者隐式地使用 SS 寄存器的其他内存引用（比如，MOV AX，[BP+6]或者 MOV AX，SS：[EAX+6]）。当没有足够的栈空间分配给局部变量时，ENTER 指令也会产生这个异常。
- 在试图加载 SS 寄存器的时候监测到段不存在。这种违例发生在栈切换期间，CALL 指令调用不同的特权级，返回到不同的特权级，一个 LSS 指令，或者对 SS 寄存器操作的 MOV 或者 POP 指令。

通过扩展栈段的界限（是界限违例的情况下）或者重新加载消失的栈段到内存（是段不存在违例）可以恢复这个故障。

异常错误码：如果异常是跨特权级调用期间出现的栈段不存在或者新栈溢出引起的，则错误码中包含指向引起异常的段的选择子。这里，异常处理程序会检测选择子所指段的描述符的存在标志来确定引起异常的原因。对于正常的界限违例（对于在使用中的栈段来说），错误码设为 0。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容通常指向产生异常的指令。但是，当异常是由于在任务切换期间加载不存在段引起的时候，CS 寄存器和 EIP 寄存器指向新任务的第一条指令。

进程状态变化：栈故障异常不改变进程的状态，因为产生故障的指令没有执行。这里，在异常处理程序纠正了栈故障之后，发生异常的指令可以重新执行。

如果栈故障发生在任务切换期间，则它会发生在新任务提交点（参看 6.3.“任务切换”）之后。这里，处理器在产生异常之前，从新的 TSS 中加载所有的状态信息（不做任何附加的界限、存在标志或类型检验）。栈故障处理程序不应该依赖于能够去使用那些没有产生其它异常的 CS、SS、DS、ES、FS、GS 寄存器中存放的段选择子。异常处理程序应该在重新开始新任务之前检验所有段选择子，否则的话，在此情况下随后产生的一般保护异常将使得诊断更加困难。（参看本章“10 号中断——非法 TSS 异常（#TS）”中的进程状态变化的描述。）

13 号中断——一般保护异常（#GP）

异常类型：故障。

描述：指出处理器检测到一类叫做“一般保护异常”的保护违例的情况之一。引起这种异常产生的状况包含所有不是引起其它异常（比如非法 TSS、段不存在、栈故障、页故障等异常）的保护违例。下列各种情况引起一般保护异常产生：

- 访问 CS、DS、ES、FS 或者 GS 段时，超越段界限。
- 引用描述符表时（任务切换和栈切换期间除外），超越段界限。
- 转移执行到一个不可执行的段。
- 写一个代码段或一个只读数据段。
- 读一个只执行代码段。
- 加载一个指向只读段的段选择子到 SS 寄存器中（除非段选择子来自任务切换期间的 TSS，此时会发生非法 TSS 异常）。
- 加载指向系统段的段选择子到 SS、DS、ES、FS 或 GS 寄存器中。
- 加载指向只执行代码段的段选择子到 DS、ES、FS 或 GS 寄存器中。
- 加载一个指向可执行段的段选择子或者空段选择子到 SS 寄存器中。
- 加载一个指向数据段的段选择子或者空段选择子到 CS 寄存器中。
- 使用包含空段选择子的 DS、ES、FS 或 GS 寄存器访问内存。
- 调用或者跳转到一个 TSS 期间，切换到忙的任务。
- 在一个没有 IRET 的任务切换期间，使用一个指向当前 LDT 的 TSS 描述符的段选择子。TSS 描述符只能放置在 GDT 中。这种状况在一个 IRET 任务切换期间会引起#TS 异常。

- 违反第 4 章“保护”中描述的特权规则。
- 超越指令 15 个字节的长度界限（这条只会发生在指令之前有冗余的前缀）。
- 加载一个 PG 标志置位（开启分页）和 PE 标志清除（关闭保护）的值到 CR0 寄存器中。
- 加载一个 NW 标志置位和 CD 标志清除的值到 CR0 寄存器中。
- 引用 IDT 表中一个非中断门、陷阱门或任务门的表项。
- 试图通过中断门或者陷阱门访问一个虚拟 8086 模式下的代码段 DPL 大于等于 0 的处理程序。
- 试图写 1 到 CR4 寄存器中的保留位。
- 试图在 CPL 不等于 0 的时候执行特权指令（参看 4.9.“特权指令”）。
- 写 MSR 中的保留位。
- 访问包含空段选择子的门。
- 在 CPL 大于引用的中断门、陷阱门或任务门的 DPL 时，执行 INT n 指令。
- 中断门、陷阱门或任务门中的段选择子没有指向一个代码段。
- LLDT 指令的段选择子操作数是局部类型（TI 置位）或者没有指向一个 LDT 类型的段描述符。
- LTR 指令的段选择子操作数是局部类型的或者指向一个不可用的 TSS。
- 调用、跳转或返回指令的代码段选择子目标操作数是空的。
- 在 CR4 寄存器的 PAE 或者 PSE 标志置位时，处理器测试到页目录指针表项的有保留位置 1。这些位是在写 CR0、CR3、CR4 等寄存器引起重新加载页目录指针表项时检测的。
- 试图写一个非零值到 MXCSR 寄存器。
- 在 SSE/SSE2/SSE3 指令需要 16 字节对齐时，执行这些指令访问一个非 16 字节对齐的内存地址。这种情况也适用于栈段。

一般保护异常之后，进程或者任务可以重新开始。如果异常发生在调用异常处理程序的时候，则被中断的进程可以重新开始，但是中断将丢失。

异常错误码：处理器把一个错误码压入异常处理程序的栈中。如果是在加载段描述符的时候检测到故障条件的，则错误码包含指向描述符的段选择子或者指向 IDT 的向量号。否则的话，错误码是 0。错误码中选择子的来源可能有下面这么几个：

- 指令的操作数。

- 来自于指令操作数的门中的选择子。
- 任务切换中的 TSS 中的选择子。
- IDT 向量号。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向产生异常的指令。

进程状态变化：一般情况下，一般保护异常不改变进程的状态，因为非法指令或者操作没被执行。可以设计异常处理程序来纠正一般保护异常引起的所有状况，并且在不失连续性的情况下重新开始进程或者任务。

如果一般保护异常发生在任务切换期间，它可以发生在新任务提交点（参看 6.3.“任务切换”）的前或者后。如果发生在任务提交点之前，进程状态不改变。如果发生在提交点之后，处理器将在产生异常之前，从新的 TSS 中加载所有状态信息（不做额外的界限、存在标志或者类型检验）。一般保护异常处理程序不应该依赖于能够去使用那些没有产生其它异常的 CS、SS、DS、ES、FS 和 GS 寄存器中存放的段选择子。（参看本章“10 号中断——非法 TSS 异常（#TS）”中的进程状态变化的描述。）

14 号中断——页故障异常（#PF）

异常类型：故障。

描述：指出在开启分页（CR0 寄存器的 PG 标志置位）的情况下，处理器使用页转换机制把线性地址转换为物理地址的过程中检测到下列状况之一：

- 页目录表项或页表项的用于地址转换的 P（存在）标志被清除，表明包含操作数的页表或者页不在内存。
- 例程没有足够的特权访问标明的页（也就是，运行于用户态的例程试图访问管理态的页）。
- 运行于用户态的代码试图写一个只读页。对于 Intel 486 及后来的处理器，如果 CR0 的 WP 标志置位，则运行于管理态的代码如果去写用户台的只读页，也会触发页故障。
- 一个或多个页目录表项的保留位置 1。参看下面关于 RSVD 错误码标志的描述。

异常处理程序可以从页不存在状况下恢复并且可以在不失连续性的情况下重新开始进程或者任务。也可以在特权违例之后重新开始，但是，引起特权违例的问题可能不可修复。

异常错误码：有的（特殊格式）。处理器向页故障处理程序提供两个信息项来帮助诊断异常并恢复它：

- 一个栈中的错误码。页故障的错误码不同于其它异常（参看图 5-7）。错误码告诉异常处理程序四件事：

  ——P 标志表明异常是由于一个不存在页（0）还是访问权限违例或是使用了保留位（1）。

  ——W/R 标志表明引起异常的内存访问是读（0）还是写（1）。

  ——U/S 标志表明异常发生时处理器是在用户态（1）执行还是在管理态（0）执行。

  ——RSVD 标志表明处理器在页目录的保留位中检测到了 1，此时控制寄存器 CR4 的 PSE 或者 PAE 标志都置为 1。(PSE 标志只在 Pentium 4、Intel Xeon、P6 系列和 Pentium 等处理器中可用,PAE 标志只在 Pentium 4、Intel Xeon、P6 系列等处理器中可用，在早期的 IA-32 处理器中，RSVD 标志位是保留的。)

- CR2 寄存器的内容。处理器把产生异常的 32 位线性地址加载到 CR2 寄存器中。页故障处理程序可以利用这个地址来确定相应的页目录表项和页表项。在页故障处理程序中也可能发生另外一次页故障。处理程序应该在第二次页故障可能发生之前保存 CR2 寄存器的内容[1]。如果页故障是由于页级保护引起的，则当故障发生时，置位页目录表项的访问位。IA-32 处理器如何解释相应页目录表项中的访问位与模型相关，而且在架构上也没有定义。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容通常指向产生异常的指令。如果页故障异常发生在任务切换期间，则 CS 寄存器和 EIP 寄存器可能指向新任务的第一条指令（参看下面“进程状态变化”的描述）。

进程状态变化：正常情况下页故障异常不改变进程状态，因为引起异常的指令未被执行。页故障异常处理程序纠正了违例（比如把消失的页调入内存）之后，进程或者任务的执行可以重新开始。

当页故障异常发生在任务切换期间时，进程状态可能会变化，具体如下所述。任务切换的时候，页故障异常可能会发生在下列操作中：

[1] 一旦检测到页故障处理器就修改 CR2 寄存器的内容。如果第二次页故障发生在较早的页故障正在调度当中，则发生故障的线性地址将重写 CR2 寄存器的内容（替换掉以前的地址）。即使页故障导致一个双故障或者发生在双故障的调度过程当中，也要重写 CR2 寄存器的。

- 写原进程状态到任务的 TSS 中的时候。
- 读 GDT 来确定新任务的 TSS 描述符的时候。
- 读新任务的 TSS 的时候。
- 从新任务中用段选择子读相关的段描述符的时候。
- 读新任务的 LDT 来验证保存在 TSS 中的段寄存器的时候。

最后两种情况下，异常发生在新任务的场境中。指令指针指向新任务的第一条指令，而不是导致任务切换的指令（或者是中断的情况下最后一条被执行的指令）。如果操作系统的要求是允许任务切换期间发生页故障，则页故障处理程序应该通过任务门调用。

如果页故障发生在任务切换期间，则处理器将在异常产生之前，从新的 TSS 中加载全部的状态信息（不执行额外的界限、存在标志或者类型检验）。因此，页故障异常处理程序不应该依赖于能够去使用那些没有产生其它异常的 CS、SS、DS、ES、FS 和 GS 寄存器中存放的段选择子。（参看本章“10 号中断——非法 TSS 异常（#TS）”中的进程状态变化的描述。）

附加的异常处理信息：应该特别小心去确保发生在显式的栈切换期间的异常不引起处理器去使用非法栈指针（SS:ESP）。为 16 位 IA-32 处理器写的软件通常使用一对指令去切换到新栈，比如：

```
MOV SS，AX
MOV SP，StackTop
```

当在 IA-32 处理器上执行这个代码时，有可能在段选择子加载到 SS 寄存器之后，EIP 寄存器还未加载之前，获取页故障、一般保护异常或者对齐异常。此时，栈指针的两个部分（SS 和 ESP）是不一致的，用了新栈段和旧的栈指针。

如果异常处理程序切换到一个定义好的栈（也就是，处理程序是一个任务或者特权级更高的例程），则处理器不使用不一致的栈。但是，如果异常处理程序在相同特权级下调用并且来自同一个任务，则处理器将试图使用不一致的栈指针。

在发生故障的任务（用陷阱门或者中断门）中处理页故障、一般保护异常或者对齐异常的系统中，和异常处于同一特权级的执行软件应该用 LSS 指令而不是用前面所述的一对 MOV 指令来初始化一个新栈。当异常处理程序运行在特权级 0 时（正常情况），问题就局限于运行在特权级 0 的例程或者任务，典型情况下就是操作系统内核。

16 号中断——x87 FPU 浮点错误（#MF）

异常类型：故障。

描述：指出 x87 FPU 检测到一个浮点错误。CR0 寄存器的 NE 标志必须置位，16 号中断浮点错误异常才可能产生。（参看 2.5.“控制寄存器”有关 NE 标志的描述。）

注意

SIMD 浮点异常（#XF）是通过 19 号中断发信号的。

当执行 x87 FPU 指令时，x87 FPU 检测并报告下述 6 类浮点错误异常：

- 非法操作（#I）

  ——栈溢出或者下溢（#IS）

  ——非法算术操作（#IA）
- 被零除（#Z）
- 非规格化的操作数（#D）
- 数值溢出（#O）
- 数值下溢（#U）
- 不精确结果（精度型）（#P）

以上每一种出错情况都代表 x87 FPU 的一种异常类型。对每一种异常类型，x87 FPU 都在 x87 FPU 状态寄存器中提供一个标志位，在 x87 FPU 控制寄存器中提供一个屏蔽位。如果 x87 FPU 检测到一个浮点错误并且相应异常类型的屏蔽位置位，则 x87 FPU 自动通过默认方式（预定义好的）对此进行相应和处理，之后继续程序的执行。对于大多数浮点应用程序来说，默认响应设计成向它提供一个合理的结果。

如果针对某类异常的屏蔽位被清除，而 CR0 的 NE 标志置位，则 x87 FPU 进行如下几步操作：

1．设置 FPU 状态寄存器中必要的标志位。

2．等待直到在指令流中遇到下一个“等待中的”x87 FPU 指令或者遇到 WAIT/FWAIT 指令。

3．产生一个内部错误信号，引起处理器产生一个浮点异常（#MF）。

在执行一个等待中的 x87 FPU 指令或者 WAIT/FWAIT 指令之前，x87 FPU 检验待处理的 x87 FPU 浮点异常（正如前面第 2 步描述的）。如果指令是 FNINIT、FNCLEX、FNSTSW AX、FNSTCW 和 FNSAVE，则待处理的 x87 FPU 浮点异常被忽略为“无等待的”x87 FPU

指令。当执行状态管理指令 FXSAVE 和 FXSTOR 时，待处理的 x87 FPU 浮点异常也被忽略。

所有的 x87 FPU 浮点错误状况都可以被恢复。x87 FPU 浮点错误异常处理程序可以从 x87 FPU 状态字中的标志位的设置来确定错误状况。参看《第 1 卷：基础构架》中的第 8 章中的“软件异常处理”。

异常错误码：无。x87 FPU 为自己提供错误信息。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向浮点产生时将要执行的符点指令或者 WAIT/FWAIT 指令。这个不是发生故障的指令，从它这里检测到错误条件的。发生故障指令的地址包含在 x87 FPU 指令指针寄存器中。参看《第 1 卷：基础构架》中的第 8 章中的“x87 FPU 指令和操作数（数据）指针”。

进程状态变化：通常进程的状态会随着 x87 FPU 浮点异常而变化，因为异常的处理会延迟到故障之后等待中的 x87 FPU 浮点指令或者 WAIT/FWAIT 指令出现。但是，x87 FPU 保存了足够的有关错误状况的信息，如果需要的话可以从错误中恢复并重新开始发生故障指令的执行。

在非 x87 FPU 浮点指令依赖于 x87 FPU 浮点指令的结果的地方，可以把 WAIT/FWAIT 指令插入到以来指令之前，以强制在依赖指令执行之前待处理的 x87 FPU 浮点异常得到处理。参看《第 1 卷：基础构架》中的第 8 章中的“x87 FPU 异常同步”。

17 号中断——对齐检验异常（#AC）

异常类型：故障。

描述：指出当开启对齐检验时，处理器检测到一个没对齐的内存操作数。对齐检验只在数据（栈）段（不在代码或者系统段）中执行。对齐检验违例的一个例子就是把字保存在奇地址或者双字保存在非 4 的整数倍的地址。表 5-7 列出了处理器识别的各种数的对齐要求。

表 5-7 数据类型的对齐要求

数据类型	地址必须被整除
字	2
双字	4
单精度符点（32 位）	4
双精度符点（64 位）	8

双扩展精度符点（80 位）	8
四字	8
双四字	16
段选择子	2
32 位远指针	2
48 位远指针	4
32 位指针	4
GDTR、IDTR、LDTR 或者任务寄存器的内容	4
FSTENV/FLDENV 保存区域	4 或 2，取决于操作数大小
FSAVE/FRSTOR 保存区域	4 或 2，取决于操作数大小
位串	2 或 4，取决于操作数大小的属性

注意，对齐检验异常（#AC）只为数据类型必须对齐到字、双字、四字的边界上产生。如果 128 位数据类型没有对齐到一个 16 字节的边界上，则产生一般保护异常(#GP)。

下列条件必须都具备才能开启对齐检验：

- CR0 寄存器的 AM 标志置位。
- EFLAGS 寄存器的 AC 标志置位。
- CPL 是 3（保护模式或者虚拟 8086 模式）。

只有当系统运行在 CPL 为 3 的特权级上（用户态）才会产生对齐检验异常（#AC）。默认属于特权级 0 的内存引用如加载段描述符不产生对齐检验异常，尽管同样情况在特权级 3 下的内存引用会产生对齐检验异常。

在特权级 3 下保存 GDTR、IDTR、LDTR 和任务寄存器的内容会产生对齐检验异常。虽然正常情况下应用程序不保存这些寄存器，但是，通过调整保存在偶字地址的信息可以避免这个故障。

FXSAVE 和 FXRSTOR 指令保存和恢复 512 个字节的数据结构，其第一个字节必须是 16 字节对齐的。如果执行这些指令（CPL 为 3）时对齐检验异常（#AC）是开启的，则根据 IA-32 处理器的不同，一个没对齐的内存操作数会引起或者对齐检验异常或者一般保护异常。(参看《第 2 卷：指令集参考》中的第 3 章中的“FXSAVE——保存 x87 FPU、MMX、SSE 和 SSE2 的状态”和“FXRSTOR——恢复 x87 FPU、MMX、SSE 和 SSE2 的状态”。)

MOVUPS 和 MOVUPD 指令执行 128 位不对齐加载和保存，当它们的操作数不是 16 字节边界对齐的时候，不产生一般保护异常（#GP)。但是，如果开启对齐检验，则会 2、4 和 8 字节不对齐将会被检测到并产生一个对齐检验异常。

FSAVE 和 FRSTOR 指令产生不对齐饮用，将会引起对齐检验故障。应用程序几乎不用这些指令。

异常错误码：有（总是零）。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向产生异常的指令。

进程状态变化：进程的状态不随着对齐检验故障而改变，因为指令还未被执行。

18 号中断——机器检验异常（#MC）

异常类型：终止。

描述：指出处理器检测到一个内部机器错误或者总线错误，或者外部代理检测到一个总线错误。机器检验异常是模型相关的，只在 Pentium 4、Intel Xeon、P6 系列和 Pentium 等处理器中可用。而 Pentium 4、Intel Xeon、P6 系列和 Pentium 等处理器中机器检验异常的具体实现也是不同的，这些实现可能与未来的 IA-32 处理器不兼容。（使用 CPUID 指令来确定这个特征是否存在。）

外部代理检测到的总线错误是通过特定引脚发信号给处理器的，这些引脚是：Pentium 4、Intel Xeon、P6 系列等处理器的 BINIT#引脚和 MCERR#引脚，Pentium 处理器的 BUSCHK#引脚。当这些引脚之一开启时，激活引脚将引起错误信息加载进机器检验寄存器并产生机器检验异常。

机器检验异常和机器检验构架将在第 14 章“机器检验构架”中详细讨论。也可以参看各个处理器数据参考手册了解有关硬件信息。

异常错误码：无。错误信息由机器检验 MSR 提供。

保存的指令指针：对 Pentium 4 和 Intel Xeon 等处理器而言，保存的扩展机器检验状态寄存器直接与引起机器检验异常产生的错误相关（参看 14.3.1.3.“IA32_MCG_STATUS MSR”和 14.3.2.5.“IA32_MCG 扩展机器检验状态 MSR”）。

对 P6 系列处理器而言，如果 MCG_STATUS_ MSR 的 EIPV 标志置位，则保存的 CS 寄存器和 EIP 寄存器直接与引起机器检验异常产生的错误相关。如果该标识位被清除，则保存的指令指针不与错误相关（参看 14.3.1.13. “IA32_MCG_STATUS MSR”）。

对 Pentium 处理器而言，保存的 CS 和 EIP 等寄存器可能不与错误相关。

进程状态变化：机器检验机制通过设置 CR4 寄存器的 MCE 标志来开启的。

对 Pentium 4、Intel Xeon、P6 系列和 Pentium 等处理器而言，进程状态随机器检验异常而变化，并且产生一个终止异常。对终止异常而言，关于异常的信息可以从机器检验 MSR 中收集，但是，进程通常不能重新开始。

如果机器检验异常是关闭的（CR4 的 MCE 标志清除），机器检验异常使得处理器进入宕机状态。

19 号中断——SIMD 符点异常（#XF）

异常类型：故障。

描述：指出处理器检测到 SSE/SSE2/SSE3 SIMD 符点异常。MXCSR 寄存器的相应标志位必须置位，并且这个中断的特定未屏蔽异常将产生。

在执行 SSE/SSE2/SSE3 SIMD 符点指令的时候，有可能产生 6 类数值异常状况：

- 非法操作（#I）
- 被零除（#Z）
- 非正规的操作数（#D）
- 数值溢出（#O）
- 数值下溢（#U）
- 不精确结果（精度型）（#P）

非法操作、被零除和非正规操作数异常是计算前异常。也就是，它们是在算术异常发生之前被检测到的。数值下溢、数值溢出和不精确结果异常是计算后异常。

参看《第 1 卷：基础架构》的第 11 章中的“SIMD 符点异常”。

当 SIMD 符点异常产生时，处理器从事下列两件事情中的一种：

- 通过产生最合理的结果和允许进程不被打断地继续执行来自动处理异常。这是对屏蔽的异常的响应。
- 产生一个 SIMD 符点异常，进而调用软件异常处理程序。这是对未屏蔽的异常的响应。

6 种 SIMD 符点异常情况都在 MXCSR 寄存器中有相应的标志位和屏蔽位。如果一种异常被屏蔽了（MXCSR 寄存器中相应的屏蔽位置位），则处理器自动采取默认的操作并继续计算。如果一种异常未被屏蔽（MXCSR 寄存器中相应的屏蔽位清除），并且操作系统支持 SIMD 符点异常（CR4 寄存器中的 OSXMMEXCPT 标志置位），则通过 SIMD 异常调用一个软件异常处理程序。如果异常未被屏蔽并且 OSXMMEXCPT 标志（表明操作系统不支持 SIMD 符点异常），则发出非法操作码异常（#UD）信号而不是 SIMD 符点异常。

注意，因为 SIMD 符点异常是精确的并且及时发生的，所以，这种情况不会出现在

一个 x87 FPU 指令、一个 WAIT/FWAIT 指令或者另外一个 SSE/SSE2/SSE3 指令捕获一个待处理的未屏蔽的 SIMD 符点异常的地方。

在一个 SIMD 符点异常发生的同时，如果该 SIMD 符点异常是被屏蔽的（引起相应的异常标志被置位）并且随后该 SIMD 符点异常又不屏蔽了，那么，当该异常不屏蔽时，就不产生异常。

当 SSE/SSE2/SSE3 SIMD 符点指令对组合操作数（有 2 个或 4 个子操作数组成）进行操作时，有可能检测到多个 SIMD 符点异常条件。如果只是检测到一个针对一个或多个子操作数集的异常条件，则为每一个检测到的异常条件都设置异常标志位。比如，检测到的针对一个子操作数的非法异常将不会阻止针对另一个子操作数的被零除异常的报告。但是，当针对一个子操作数的两个或多个异常条件产生时，只报告一个异常，优先关系参看表 5-8。这个优先关系有时会导致较高优先级的异常条件被报告了而较低优先级的异常条件被忽略了。

表 5-8 SIMD 符点异常优先级

优先级	描　　述
1（最高）	由于 SNaN 操作数造成的非法操作异常（或者任何针对最大、最小或某些比较和转换操作的 NaN 操作数）。
2	QNaN 操作数 [1]。
3	前面没有提及的任何其它非法操作异常或者被零除异常 [2]。
4	非正规操作数异常 [2]。
5	数值溢出或者下溢可能连同非精确结果异常 [2]。
6（最低）	非精确结果异常 [2]。

备注：

1．尽管 QNaN 本身不是异常，但是，一个 QNaN 操作数的处理有比较低优先级异常高的优先权。比如，一个 QNaN 被零除导致一个 QNaN，而不是一个被零除异常。

2．如果屏蔽了，则指令流执行继续，并且同时可能发生一个较低优先级的异常。

异常错误码：无。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向当 SIMD 符点异常产生时的 SSE/SSE2/SSE3 指令。这个就是检测到错误条件时发生故障的指令。

进程状态变化：进程的状态不随着 SIMD 异常而改变，因为除非这个特定异常被屏蔽了，否则会被立即处理的。提供的状态信息通常足够用来从错误中恢复，并且如果需要，发生故障的指令可以重新执行。

32-255 号中断——未定义中断

异常类型：待定。

描述：指出处理器做了下列事情之一：

- 执行了一个 INT n 指令，n 为向量号 32 到 255 之间的任何一个。
- 响应 INTR 引脚或者来自本地 APIC 的中断请求，而这些中断与中断向量号 32 到 255 之间进行了绑定。

异常错误码：待定。

保存的指令指针：保存的 CS 寄存器和 EIP 寄存器的内容指向 INT n 指令之后的第一条指令或者 INTR 信号发生处的后一条指令。

进程状态变化：INT n 指令或者 INTR 信号不改变进程的状态。INT n 在指令流中产生一个中断。当收到 INTR 信号时，处理器会提交这之前所有的状态，然后才会响应中断。因此，在从中断处理程序返回时，进程执行可以重新开始。

第 6 章 任务管理

这一章描述 IA-32 架构下的任务管理工具。这些工具只有当处理器运行于保护模式下时才是可用的。

6.1. 任务管理概述

一个任务就是一个工作单元，处理器可对其进行调度、执行和挂起。任务可被用来执行一个程序、一个进程、一个操作系统服务例程、一个中断或异常处理例程，或一个内核或管理程序的实用程序。

IA-32 架构提供了一套机制，用以保存任务的状态、调度执行任务和在任务之间进行切换。在保护模式下，处理器的所有执行都发生在一个任务内。即使是简单的系统，也至少要定义一个任务。更复杂一些的系统则能够利用处理器提供的任务管理工具对多任务应用提供支持。

6.1.1. 任务结构

一个任务由两部分构成：任务执行空间和任务状态段（TSS）。任务执行空间则由代码段、栈段、一个或多个的数据段组成（参考图 6-1）。若操作系统或管理程序使用了处理器的特权级保护机制，则执行空间还要为每一特权级提供一个独立的栈。

TSS 指定组成任务执行空间的各个段，提供存储任务状态信息的空间。在多任务系统中，TSS 还提供任务的链接机制。

注意

本章主要描述 32 位的任务和 32 位的 TSS 结构。参考 6.6.“16 位任务状态段(TSS)”，以获取有关 16 位任务和 16 位 TSS 结构的信息。

任务由指向其 TSS 的段选择子标识。当任务被加载进处理器执行时，则把它的 TSS 的段选择子、基地址、界限以及段描述符的属性等加载到任务寄存器中（参考 2.4.4.“任务寄存器（TR)”)。

如果任务使用了分页机制，则把其使用的页目录表的基地址加载到控制寄存器 CR3

中。

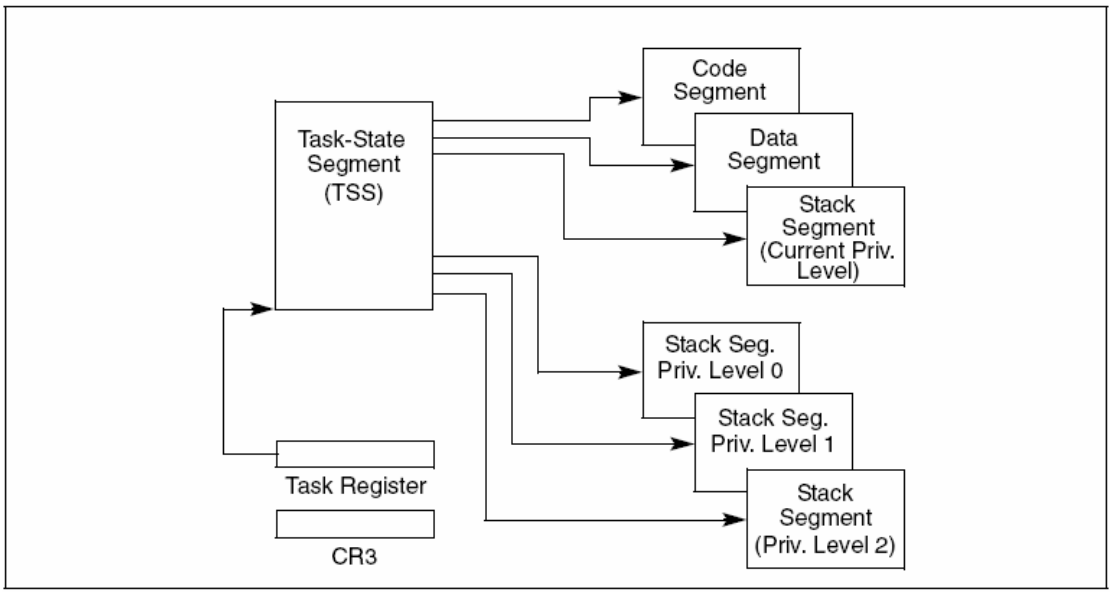

图 6-1 任务结构

6.1.2.任务状态

下列各项定义当前执行任务的状态：

- 任务的当前执行空间，由段寄存器（CS、DS、SS、ES、FS 和 GS）中的段选择子指定。
- 通用寄存器的状态。
- EFLAGS 寄存器的状态。
- EIP 寄存器的状态。
- 控制寄存器 CR3 的状态。
- 任务寄存器的状态。
- LDTR 寄存器的状态。
- I/O 位图基地址和 I/O 位图（位于 TSS 中）。
- 特权级 0、特权级 1 和特权级 2 的栈指针（位于 TSS 中）。
- 指向前一个执行过的任务的链接（位于 TSS 中）。

任务调度之前，除了任务寄存器的状态之外，所有上面这些项都保存在将要被调度的任务的 TSS 里。另外，TSS 中不包含完整的 LDTR 寄存器的内容，只有 LDT 的段选择子。

6.1.3.执行任务

软件或处理器可以用以下方式中的任何一种来调度任务执行：

- 用 CALL 指令显式地调用任务。
- 用 JMP 指令显式地跳转到任务。
- （由处理器）隐式地调用中断处理程序任务。
- 隐式地调用异常处理程序任务。
- EFLAGS 寄存器的 NT 标志置位时的任务返回（由 IRET 指令发出）。

所有这些调度方法都是用指向任务门或任务 TSS 的段选择子来识别被调度的任务的。当用 CALL 或 JMP 指令调度任务时，指令中的选择子可以直接指向 TSS，也可以指向包含 TSS 选择子的任务门。通过调度任务来处理中断或异常时，相应中断或异常对应的 IDT 项必须包含一个任务门，门中含有指向中断或异常处理程序任务的 TSS 的选择子。

一个任务被调度执行时，会自动地在当前任务和被调任务之间发生任务切换。切换时，当前任务的执行环境（称做任务状态或**场境**（context））保存进其 TSS 中，该任务随即被挂起。然后，处理器加载被调任务的场境到各个寄存器中，从刚加载的 EIP 寄存器指向的指令处开始执行新任务。若该任务是系统上次初始化以来的首次执行，则 EIP 指向任务代码的第一条指令；否则，指向任务上次被挂起时执行的最后一条指令的下一条指令。

如果是由当前任务（调用任务）调用了被调度的任务（被调任务）而发生了任务切换，则把调用任务的 TSS 的选择子保存到被调任务的 TSS 中，以提供返回调用任务的链接。

所有 IA-32 架构的处理器都不允许任务递归。一个任务不能调用或跳转到它自身。

可通过切换任务到处理程序任务来处理中断和异常。此种情况下，处理器不仅可以执行任务切换来处理中断或异常，还可以在从中断或者异常处理程序任务返回时自动切换回被中断的任务。这一机制可用来处理在中断任务执行期间发生的中断。

作为任务切换的组成部分，处理器还可以切换到另一个 LDT 上，即允许每个任务拥有一个不同的基于 LDT 段的从逻辑到物理地址的映射机制。任务切换时也会重新装载页目录表基地址寄存器（CR3），允许每个任务拥有自己的一套页表。这些保护工具帮助隔离任务，防止它们相互干扰。如果这些保护机制中的一项或多项未启用，则处

理器就无法提供任务间的保护。即使对于使用多特权级保护的操作系统也是如此。比如，如果运行于特权级 3 上的多个任务使用相同的 LDT 和页表，则彼此之间可以互访代码，破坏数据和栈。

处理多任务应用也可以不使用处理器提供的任务管理工具，而是使用软件来处理，即把每一个软件定义都成在单个 IA-32 架构任务的场境中执行的任务。

6.2.任务管理数据结构

处理器定义了 5 种数据结构来处理与任务相关的活动：

- 任务状态段（TSS）。
- 任务门描述符。
- TSS 描述符。
- 任务寄存器。
- EFLAGS 寄存器的 NT 标志。

当工作于保护模式时，要为至少一个任务创建 TSS 和 TSS 描述符，并把它的 TSS 的段选择子加载到任务寄存器中（使用 LTR 指令）。

6.2.1.任务状态段（TSS）

恢复任务所需的处理器状态信息保存在一个称做任务状态段（TSS）的系统段中。图 6-2 显示了为 32 位 CPU 设计的任务 TSS 的格式。（为了和 16 位的 Intel 286 处理器任务兼容，还提供了另外一种 TSS，见图 6-9。）TSS 的域分为两部分：动态域和静态域。

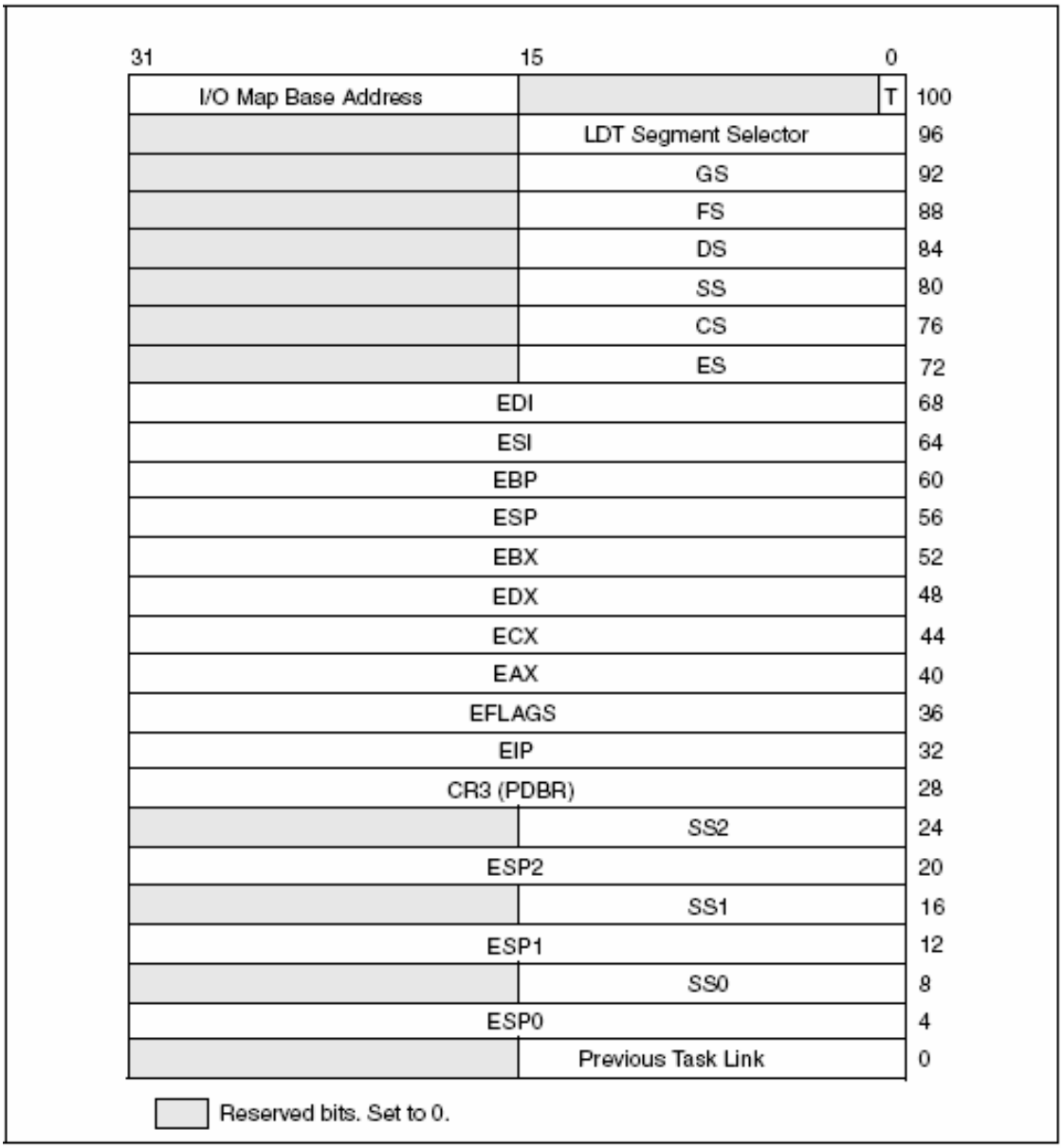

图 6-2 32 位任务状态段（TSS）

任务切换时，处理器自动更新被挂起任务的动态域。下面是动态域：

通用寄存器域

任务切换前 EAX、ECX、EDX、EBX、ESP、EBP、ESI 和 EDI 等寄存器的状态。

段选择子域

任务切换前保存在 ES、CS、SS、DS、FS 和 GS 等寄存器中的段选择子。

EFLAGS 寄存器域

任务切换前 EFLAGS 寄存器的状态。

EIP（指令指针）域

任务切换前 EIP 寄存器的状态。

前一个任务链接域

包含前一个任务的 TSS 的段选择子（在由调用、中断或异常引起的任务切换时更新的）。该域（常称做返回链接域）允许调用 IRET 指令切换回前一个任务。

处理器读取静态域的数据，但通常不改变它们。这些域的内容是在创建任务时设置的。下面的是静态域：

LDT 段选择子域

包含任务 LDT 的段选择子。

CR3 控制寄存器域

包含任务使用的页目录表的物理基地址。控制寄存器 CR3 也被称做页目录表基地址寄存器（PDBR）。

特权级 0、1、2 的栈指针域

这些栈指针中包含由栈段（SS0、SS1 和 SS2）的段选择子和栈中偏移（ESP0、ESP1 和 ESP2）组成的逻辑地址。注意，对某个任务来说，这些域中的值是静态的。但是，如果任务中发生栈切换，则 SS 和 ESP 的值会改变。

T（调试陷阱）标志（第 100 字节的第 0 位）

当切换到 T 标志置位的任务时，将导致处理器产生一个调试异常（参看 15.3.1.5.“任务切换异常条件”）。

I/O 位图基地址域

包含一个从 TSS 基地址到 I/O 许可位图和中断重定向位图的 16 位偏移。当这些位图存在时，它们存储在 TSS 的高地址区。I/O 位图基地址指向 I/O 许可位图的起始地址和中断重定向位图的末地址。参考《第 1 卷：基础架构》的第 13 章“输入/输出”，以获取关于 I/O 许可位图的资料（原文中此处写的是第 12 章，经查英文手册，现资料中为第 13 章，故此改正——译者）。参考 16.3.“虚拟 8086 模式的中断和异常处理”，获取中断重定向位图的详细描述。

任务转换时，处理器要读取 TSS 的部分内容（最前端的 104 字节）。如果开启了分页机制，必须注意避免使 TSS 的这部分跨越两个页面。如果确实出现这种情况，那么这两个内存页必须同时存在于内存中，且是连续的。

这种限制的原因是，在任务切换中访问 TSS 时，处理器是从 TSS 第一个字节的物理地址开始连续读取和写入 104 个字节的。如果页边界正好出现在这个里面，则在页边界处不能进行地址转换（也就是说，后面一页的线性地址与前面一页未必连续。如

果这样的话，则后面页不是前面页的后续页——译者)。于是，在访问 TSS 时，如果 104 个字节的部分内容未在内存中，或存在但不连续，处理器将访问不正确的 TSS 信息，而不产生页故障异常。此后的任务切换会因为读取这些错误信息而导致不可恢复的异常事件。

而且，如果使用了分页的话，则前一个任务的 TSS 和当前任务的 TSS 相应的页以及它们的描述符项都应该标为“读/写”。如果在任务切换之前包含所有这些数据结构的页都在内存中，则任务切换将会比较快。

6.2.2. TSS 描述符

象所有其它段一样，TSS 也是通过段描述符定义的。图 6-3 显示了 TSS 描述符的格式。TSS 描述符只能放在 GDT 中，而不能放在 LDT 或者 IDT 中。

在执行 CALL 或者 JMP 期间，试图用一个 TI 标志置位（表示在当前 LDT 中）的段选择子访问 TSS 会引起一般保护异常(#GP),并在执行 IRET 时引起非法 TSS 异常(#TS)。如果把 TSS 的段选择子加载进段寄存器，则会产生一般保护异常。

类型域中的忙(B)标志指出任务是否忙。一个忙的任务当前正在运行或者被挂起。1001B 这个类型值表明是一个不活动的任务，1011B 表明是一个忙的任务。任务自身不能递归调用。处理器使用忙标志去探测对已经被中断任务的调用企图。为确保一个任务只有一个忙标志与之相关，每个 TSS 只能有一个 TSS 描述符指向它。

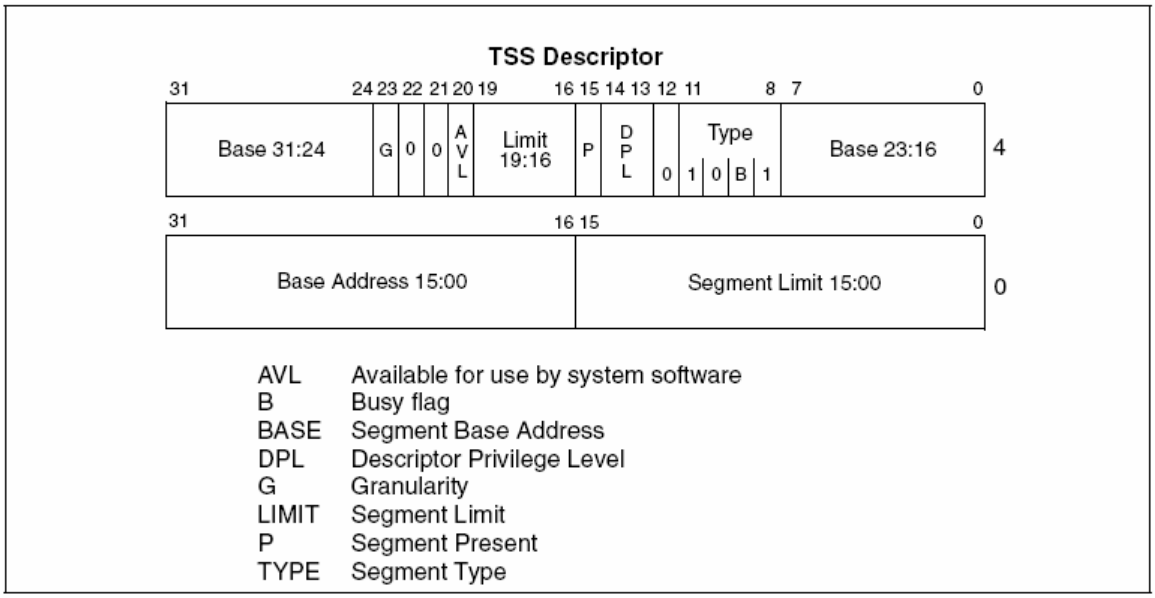

图 6-3 TSS 描述符

基地址、界限、DPL、粒度、存在标志等域的作用与它们在数据段描述符（参看 3.4.3.“段描述符”）中的用法类似。对 32 位 TSS 来说，如果 TSS 描述符中的 G 标志为 0，则界限域的值必须等于或大于 67H，也就是 TSS 最小长度少一个字节的数值。试图切换到 TSS 描述符小于 67H 的任务将产生一个非法 TSS 异常（#TS）。如果 I/O 许可位图包含在 TSS 中的话，则需要一个较大的界限。如果操作系统保存额外的数据在 TSS 中，则需要一个更大的界限。任务切换时，处理器不检验大于 67H 的界限，但是，当访问 I/O 许可位图或者中断重定向位图时，则要做检验的。

任何一个可以访问 TSS 描述符的进程或者例程（也就是它的 CPL 的数值等于或者小于 TSS 描述符的 DPL）都可以通过调用或者跳转来调度这个任务。

在大部分系统中，TSS 描述符的 DPL 应该设为比 3 小的值，因而只有特权软件可以执行任务切换。但是，在多任务应用系统中，可以把某些 TSS 描述符的 DPL 设为 3，以允许在应用程序特权级（用户态）进行任务切换。

6.2.3. 任务寄存器

任务寄存器保存 16 位段选择子和当前任务的 TSS 描述符的全部内容（32 位基地址、16 位段界限和描述符属性）。这些信息是从 GDT 中的当前任务的 TSS 描述符中复制出来的。图 6-4 显示了处理器使用任务寄存器中的信息访问 TSS 的路径。

任务寄存器包括两个部分，一部分看得见（可被软件读或者修改），一部分看不见（由处理器维护，软件不可访问）。看得见部分的段选择子指向 GDT 中的 TSS 描述符。处理器使用任务寄存器中看不见的部分来高速缓存 TSS 的段描述符。高速缓存这些值使得任务的执行更加有效率，因为，处理器不需要从内存中取这些值来引用当前任务的 TSS 了。

LTR（加载任务寄存器）和 STR（保存任务寄存器）指令用来加载和读取任务寄存器的可见部分。LTR 加载指向 GDT 中 TSS 描述符的段选择子（源操作数）到任务寄存器中，并且从 TSS 描述符中把不可见部分也加载进任务寄存器中。这是个特权指令，只有 CPL 为 0 时才能执行。LTR 指令一般用于系统初始化时期放置一个初始值在任务寄存器中。之后，任务寄存器的内容是在任务切换期间隐式的改变的。

STR 指令把任务寄存器中看得见的部分保存进通用寄存器或者内存中。这个指令可

以在任何特权级的代码中运行，用来识别当前运行任务。但是，它通常用于操作系统软件。

在加电或者处理器复位的时候，任务寄存器的段选择子和基地址设为默认值 0，界限设为 FFFFH。

6.2.4.任务门描述符

门描述符提供一种间接的、保护性的对任务的访问。图 6-5 显示了任务门描述符的格式。任务门描述符可以放置在 GDT、LDT 或者 IDT 中。

任务门描述符中的 TSS 段选择子域指向 GDT 中的 TSS 描述符。这个段选择子中的 RPL 是不用的。

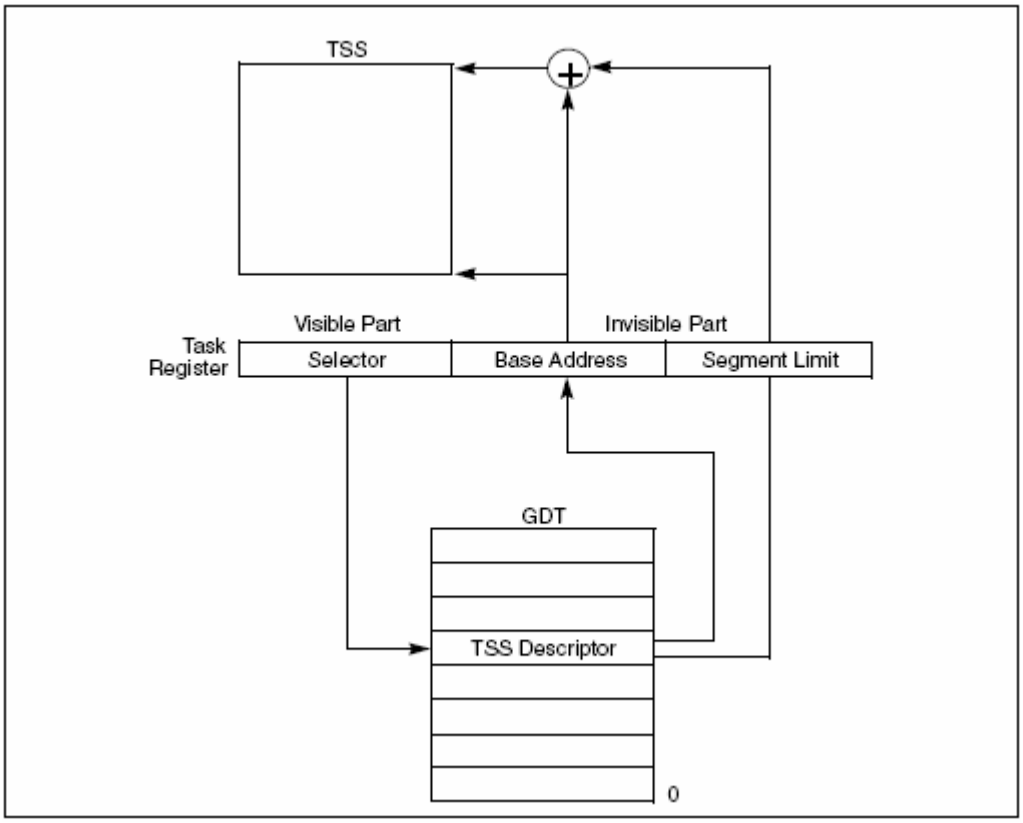

图 6-4 任务寄存器

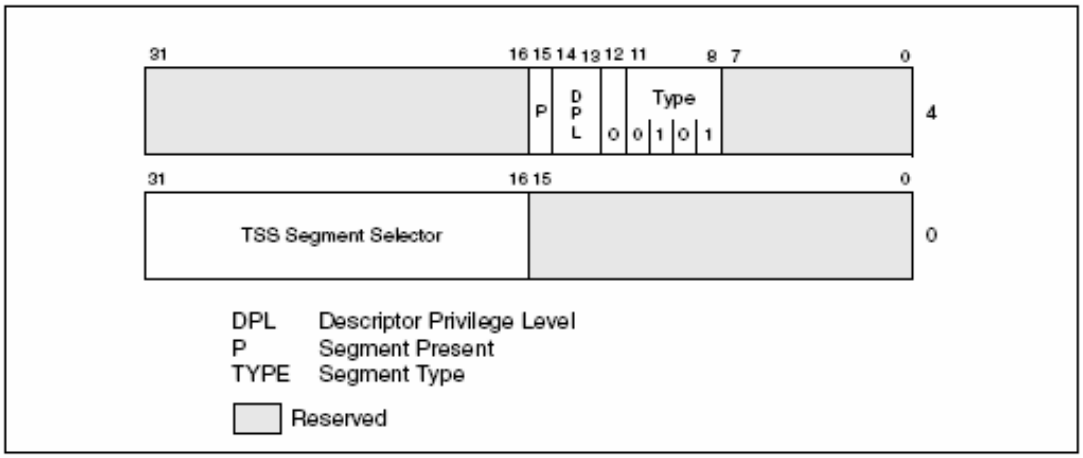

图 6-5 任务门描述符

任务门描述符中的 DPL 控制着任务切换期间对 TSS 描述符的访问。当一个进程或者例程通过任务门调用或者跳转到一个任务时，它们的 CPL 和指向这个任务门的门选择子的 RPL 域必须小于或等于这个任务门描述符的 DPL。（注意，使用任务门的时候，不使用目标 TSS 描述符的 DPL。）

只可通过任务门描述符或者 TSS 描述符来访问一个任务。这两种结构都用来满足下列需要：

- 一个任务只有一个忙标志的需要。因为任务的这个忙标志是保存在 TSS 描述符中的，而每个任务应该只有一个 TSS 描述符。但是，可能有好几个任务门指向同一个 TSS 描述符。
- 提供对任务选择性访问的需要。任务门满足了这个要求，因为它们可以放置在 LDT 中，并且有一个不同于 TSS 描述符 DPL 的 DPL。缺乏足够特权去访问 GDT 中某个任务的 TSS 描述符（它的 DPL 通常为 0）的进程或者例程，却可能通过具有较高 DPL 的任务门来访问这个任务。任务门给了操作系统较大的权限来限制对特定任务的访问。
- 通过一个独立的任务来处理中断或异常的需要。任务门可以放置在 IDT 中，这就允许用一个处理程序任务来处理中断或者异常。当中断或者异常向量指向一个任务门时，处理器就切换到指定的任务。

图 6-6 演示了 LDT 中的一个任务门、GDT 中的一个任务门和 IDT 中的一个任务门是如何都指向同一个任务的。

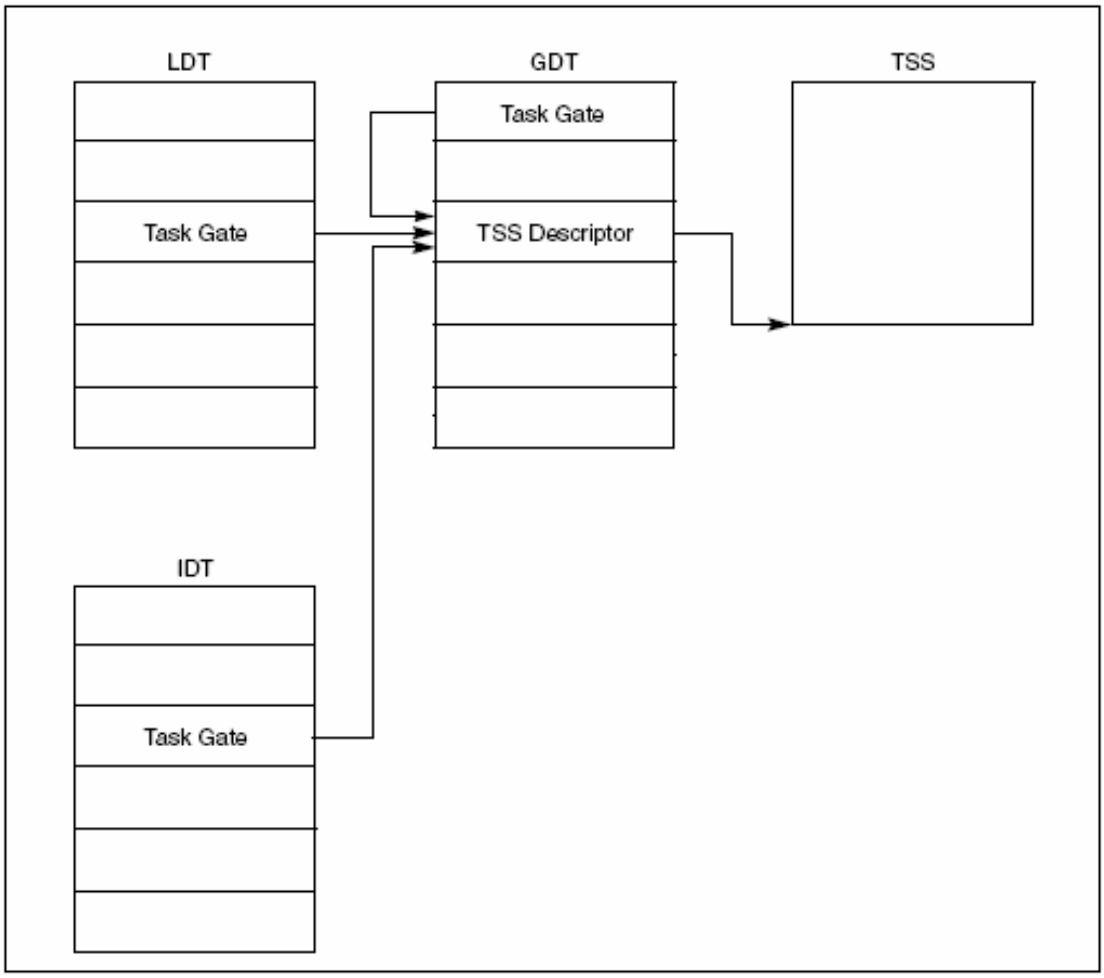

图 6-6 引用同一个任务的任务门

6.3.任务切换

下列四种方式中的任何一种都会导致处理器转移执行到另外一个任务：

- 当前进程、任务或者例程执行一个 JMP 或者 CALL 指令到 GDT 中的一个 TSS 描述符。
- 当前进程、任务或者例程执行一个 JMP 或者 CALL 指令到 GDT 或者当前 LDT 中的一个任务门描述符。
- 一个中断或者异常向量，它指向 IDT 中的一个任务门描述符。
- 当前任务执行了一个 IRET，此时 EFLAGS 寄存器中的 NT 标志是置位的。

JMP、CALL 和 IRET 指令以及中断和异常都是重定向进程的通用机制。TSS 描述符或者任务门的引用（调用或者跳转到一个任务时）或者 NT 标志的状态（执行一个 IRET 指令时）决定着任务切换是否发生。

当往一个新任务切换时，处理器执行下列操作：

1．从 JMP 或 CALL 指令的操作数、任务门、前一个任务链接域（对于 IRET 指令引起的任务切换来说）等中获取新任务的 TSS 段选择子。

2．检验当前（老）任务是否允许切换到新任务。将数据访问特权规则应用于 JMP 和 CALL 指令。当前（老）任务的 CPL 和新任务段选择子的 RPL 必须小于或等于被访问的 TSS 描述符或任务门的 DPL。允许异常、中断（INT n 指令产生的中断除外）和 IRET 指令等切换任务而不管目的任务门或 TSS 描述符的 DPL。对于 INT n 指令产生的中断，是要检验 DPL 的。

3．检验新任务的 TSS 描述符标有“存在”标志并且界限合法（大于或者等于 67H）。

4．检验新任务是可用（调用、跳转、异常或中断）或者忙（IRET 返回）的。

5．检验当前（老）TSS、新 TSS 和所有用于任务切换的段描述符都被分页到系统内存中。

6．如果是由 JMP 或者 IRET 指令发动的任务切换，则处理器清除当前（老）任务的 TSS 描述符的忙（B）标志；如果是由 CALL 指令、异常或者中断发动的，则忙（B）标志还保持置位。（参看表 6-2。）

7．如果是由 IRET 指令发动的任务切换，则处理器清除临时保存的 EFLAGS 寄存器映像中的 NT 标志；如果是由 CALL 指令、JMP 指令、异常或者中断发动的，则映像中的 NT 标志还保持不变。

8．保存当前（老）任务的状态到当前任务的 TSS 中。处理器从任务寄存器中找出当前 TSS 的基地址，然后复制下列寄存器的状态到当前 TSS 中：所有的通用寄存器、段寄存器中的段选择子、临时保存的 EFLAGS 寄存器映像和指令指针寄存器（EIP）。

9．如果是由 CALL 指令、异常或者中断发动的任务切换，处理器将设置从新任务中加载的 EFLAGS 寄存器中的 NT 标志。如果是由 IRET 指令、JMP 指令发动的，则 NT 标志将反映出从新任务中加载的 EFLAGS 寄存器中的 NT 标志的状态（参看图 6-2）。

10．如果是由 CALL 指令、JMP 指令、异常或者中断发动的任务切换，处理器将置位新任务的 TSS 描述符中的忙（B）标志；如果是由 IRET 指令发动的，则忙（B）标志保持置位。

11．用段选择子和新任务的 TSS 描述符加载任务寄存器。

12．TSS 状态加进处理器，包括：LDTR 寄存器、PDBR（控制寄存器 CR3）、EFLAGS 寄存器、EIP 寄存器、通用寄存器和段选择子。注意，加载过程中的故障会破坏架构状

态。

13．与段选择子相关的描述符被加载和考核。这个加载和考核过程中出现的任何错误都是发生在新任务的场境中。

注意

此时，如果所有的检验和保存都成功完成，则处理器提交任务切换。如果在第 1 到 11 步中出现不可恢复的错误，处理器不完成任务切换并且确保处理器回到发动任务切换那条指令执行之前的状态。如果在第 12 步出现不可恢复的错误，架构状态可能被破坏，但是会试图在前执行环境中处理错误。如果在提交点（第 13 步）之后发生不可恢复的错误，则处理器完成任务切换（不会执行额外的访问和段可用性检验）并且在开始执行新任务之前产生一个合适的异常。如果异常发生在提交点之后，则异常处理程序本身必须在开始执行新任务的执行之前完成任务切换。参看第 5 章“10 号中断——非法 TSS 异常（#TS）”。

14．开始执行新任务。（对异常处理程序来说，新任务的第一条指令好像还没被执行。）

当任务成功切换时，当前运行任务的状态总是被保存了。如果任务是重新执行的，则从保存的 EIP 的值所指的指令开始执行，寄存器都恢复成任务挂起时候的拥有值。

当切换任务时，新任务不继承挂起任务的特权级。新任务在 CS 寄存器中的 CPL 域所指定的特权级上执行，这个值是从 TSS 中加载的。因为任务是通过它们各自的地址空间和 TSS 等隔绝开的，也因为特权规则控制着对 TSS 的访问，所以任务切换时软件不需要显式地检验特权规则。

表 6-1 显示了任务切换时处理器检验的异常条件，也显示了每种检验中探测到错误时所产生的异常以及错误码所引用的段。（表中检验的顺序是 P6 系列处理器中所用的顺序。实际顺序是与模型相关的并且在其它类型的 IA-32 处理器中可能会不同。）处理这些异常的处理程序如果试图重新加载产生异常的段选择子时，可能会引起递归调用。应该在重新加载选择子之前修复这个因素（或者多个因素中的这个首要因素）。

表 6-1 任务切换时检验的异常条件

检验的条件	异常[1]	错误码引用[2]
TSS 描述符的段选择子引用 GDT 并且在 GDT 表的界限之内。	#GP #TS（对 IRET 而言）	新任务的 TSS
TSS 描述符存在于内存中。	#NP	新任务的 TSS

TSS 描述符不忙（对调用、中断或异常发动的任务切换而言）。 TSS 描述符不忙(对 IRET 发动的任务切换而言)。 TSS 段界限大于或者等于 108(对于 32 位 TSS 而言）或者 44（对于 16 位 TSS 而言)。	#GP（对 JMP、CALL、INT 而言） #TS（对 IRET 而言） #TS	任务的后向链接 TSS 新任务的 TSS 新任务的 TSS
加载 TSS 中的值进入寄存器。		
新任务的 LDT 段选择子是合法的 [3]。 代码段的 DPL 与段选择子的 RPL 相匹配。 SS 段选择子是合法的 [2]。 栈段存在于内存中。	#TS #TS #TS #SF	新任务的 LDT 新的代码段 新的栈段 新的栈段
栈段 DPL 与 CPL 相匹配。 新任务的 LDT 存在于内存中。 CS 段选择子是合法的 [3]。 代码段存在于内存中。 栈段 DPL 与选择子 RPL 相匹配。 DS、ES、FS 和 GS 段选择子是合法的 [3]。 DS、ES、FS 和 GS 段是可读的。 DS、ES、FS 和 GS 段存在于内存中。 DS、ES、FS 和 GS 段 DPL 大于或者等于 CPL（除非这些是一致性段)。	#TS #TS #TS #NP #TS #TS #TS #NP #TS	新的栈段 新任务的 LDT 新的代码段 新的代码段 新的栈段 新的数据段 新的数据段 新的数据段 新的数据段

备注：

1．#NP 是段不存在异常，#GP 是一般保护异常，#TS 是非法 TSS 异常，#SF 是栈故障异常。

2．错误码包含一个对本栏所述段的描述符的索引。

3．如果段选择子位于相容类型的表（GDT 或者 LDT）中，占据着表段界限内的一个地址，并且指向一个相容类型的描述符（比如，仅当 CS 寄存器中的段选择子指向一个代码段描述符时才是合法的）时，则它是合法的。

一旦任务切换发生时，CR0 寄存器中的 TS（任务切换）标志就被置位。当处理器的其余部分产生浮点异常时，系统软件使用 TS 标志来协调浮点单元的操作。TS 标志指出浮点单元的场境可能不同于当前任务。参看 2.5.“控制寄存器”。

6.4.任务链接

TSS 的前一个任务链接域（有时称作“返回链接”）和 EFLAGS 寄存器的 NT 标志用于返回执行到前一个任务。NT 标志表明当前任务是否在另外一个任务执行中的嵌套执行，如果是的话，当前任务的前一个任务链接域是否保存着嵌套结构中较高层次任务的 TSS 选择子（参看图 6-7）。

当 CALL 指令、中断或者异常引起任务切换时，处理器复制当前 TSS 的段选择子到新任务的 TSS 中的前一个任务链接域中，然后置位 EFALGS 寄存器中的 NT 标志。NT 标志表明当前 TSS 的前一个任务链接域已经加载了已保存的 TSS 段选择子。如果软件使用 IRET 指令挂起一个新任务，则处理器使用前一个任务链接域中的值和 NT 标志返回到前一个任务。也就是，如果 NT 标志是置位的，则处理器执行任务切换至前一个任务链接域中指定的任务。

注意

当 JMP 指令引起任务切换时，新任务是不嵌套的。也就是，NT 标志置零，前一个任务链接域不使用。当不希望嵌套时，使用 JMP 指令来调度任务。

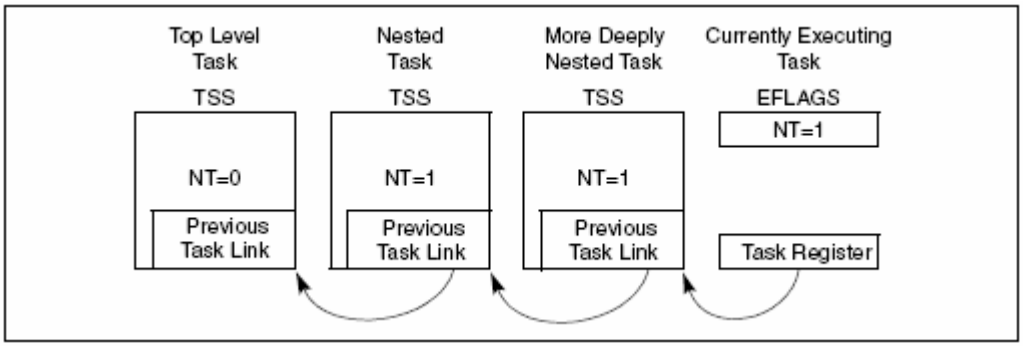

图 6-7 嵌套的任务

表 6-2 总结了任务切换期间忙标志（在 TSS 段描述符中）、NT 标志、前一个任务链接域和 TS 标志（在控制寄存器 CR0 中）等的用法。注意，NT 标志可被运行在任何特权级上的软件修改。进程设置自己的 NT 标志，然后执行一个 IRET 指令去调用当前任务 TSS 的前一个任务链接域指定的任务，这是有可能的。为防止成功地进行虚假的任务切换，操作系统应该把它创建的每个 TSS 的前一个任务链接域初始化为 0。

表 6-2 任务切换对忙标志、NT 标志、前一个任务链接域和 TS 标志的影响

标志或域	JMP 指令的影响	CALL 指令或中断的影响	IRET 指令的影响
新任务的忙标志	标志置位。之前必须已经被清除了	标志置位。之前必须已经被清除了	不变。必须已经置位
老任务的忙标志	清除标志	不变。当前置位标志	标志置位
新任务的 NT 标志	置为新任务 TSS 中的值	标志置位	置为新任务 TSS 中的值
老任务的 NT 标志	不变	不变	清除标志
新任务的前一个任务链接域	不变	加载老任务 TSS 的选择子	不变

老任务的前一个任务链接域	不变	不变	不变
CR0 控制寄存器的 TS 标志	标志置位	标志置位	置位标志

6.4.1.使用忙标志防止递归任务切换

一个 TSS 只能保存一个任务的场境。因而，一旦任务被调用（调度），一次对任务的递归调用（或者再入）会导致任务的当前状态丢失。TSS 段描述符的忙标志用来阻止再入任务的切换及任务状态信息的丢失。处理器按照下列方式管理忙标志：

1．当调度一个任务时，处理器置位新任务的忙标志。

2．如果在任务切换期间，当前任务是置于嵌套链（任务切换是由 CALL 指令、中断或者异常产生的）中的，则当前任务的忙标志还保持置位。

3．当切换到一个新任务（由 CALL 指令、中断或者异常发动的）时，如果新任务的忙标志已经置位了,则处理器产生一个一般保护异常(#GP)。(如果任务切换是由 IRET 指令发动的，则不会产生异常，因为处理器期望忙标志置位。)

4．当一个任务是由于跳转到一个新任务（任务代码中发出 JMP 指令）或者任务代码中发出 IRET 指令而结束的，则处理器清除它的忙标志，返回任务到“不忙”状态。

这种情况下，通过阻止一个任务切换到自身或者嵌套链中的一个任务，处理器来阻止递归任务切换。由于多次的调用、中断或者异常，嵌套的挂起的任务链可能会变得非常长。如果任务在这个链中，忙标志会防止它被调用。

忙标志也可用于多处理器配置,因为当处理器设置或者清除忙标志时,它遵循 LOCK 协议（在总线和高速缓存中)。这种锁会避免两个处理器同时调用同一个任务。(有关多处理器应用中设置忙标志的信息，参看 7.1.2.1.“自动加锁”。)

6.4.2.修改任务链接

在单一处理器系统中，在需要从链接的任务链中删除一个任务的情况下，使用下列步骤：

1．关闭中断。

2．改变占先任务（挂起要被删除任务的任务）的 TSS 的前一个任务链接域。假定占先任务是链中要被删除任务的下一个任务（较新的任务)。改变它的前一个任务链接

域指向链中下一个最老的任务或者更老的任务（也就是被删除任务的前一个任务——译者）。

3．清除从链中删除的任务的 TSS 段描述符的忙标志。如果有超过一个任务从链中删除，则被删除的每个任务的忙标志都要清除。

4．开启中断。

在多处理器系统中，额外的同步和顺序化的操作必须加入这个过程，以确保在改变前一个任务链接域和清除忙标志的时候，TSS 和它的描述符都被锁住。

6.5.任务地址空间

任务的地址空间由任务可以访问的段构成，包括：TSS 中引用的代码、数据、栈、系统等段，以及任务代码访问的其它段。这些段被映射到处理器的线性地址空间中，进而被映射到处理器的物理地址空间中（直接映射或者通过分页）。

TSS 中的 LDT 段域可用来给每个任务一个它自己的 LDT。每个任务都有自己的 LDT 就可以做到把每个任务相关的段都放置在自己任务的 LDT 中，以便把它自己的地址空间与其它任务隔绝。

几个任务使用同一个 LDT 也是可能的。这是一个简单地、内存有效地允许几个任务相互通讯和相互控制的方法，而不会丢失整个系统的保护性屏障。

因为所有任务都访问 GDT，所以也可以创建通过这个表中的段描述符访问的共享数据段。

如果开启了分页，则 TSS 中的 CR3 寄存器（PDBR）域允许每个任务也可以有自己的用来映射线性地址到物理地址的页表集。或者，几个任务共享同一组页表集。

6.5.1.映射任务到线性和物理地址空间

用下列两种方法之一可以映射任务到线性地址空间和物理地址空间：

- 所有任务共享同一种线性——物理地址空间映射方法。当关闭分页时，这是唯一的选择。没有分页，所有的线性地址映射到相同的物理地址。当开启分页时，为所有任务使用一张页目录表也可以获得这种线性——物理地址空间映射方法。如果支持请求分页虚拟内存的话，则线性地址空间会超过可用的物理空间。
- 每个任务有它自己的映射到物理地址空间的线性地址空间。通过为每个任务使

用一个不同的页目录表可以做到这种形式的映射。因为每次任务切换时都要加载 PDBR（CR3 控制寄存器），每个任务可以有一个不同的页目录表。

不同任务的线性地址空间可以映射到完全不同的物理地址中。如果不同的页目录表项指向不同的页表，页表指向不同的物理页，那么，任务之间就不会共享任何物理地址。

不管用两种方法中的哪一种映射任务的线性地址空间，所有任务的 TSS 必须放置在一个共享的物理区域中，以便所有任务都可访问。这种映射是需要的，以便任务切换期间处理器读取和修改 TSS 的时候，TSS 的地址映射不改变。由 GDT 映射的线性地址空间也应该映射到共享物理空间区域，否则的话，GDT 的目的就没戏了。图 6-8 显示了通过共享页表，两个任务的线性地址空间是如何在物理空间中重叠的。

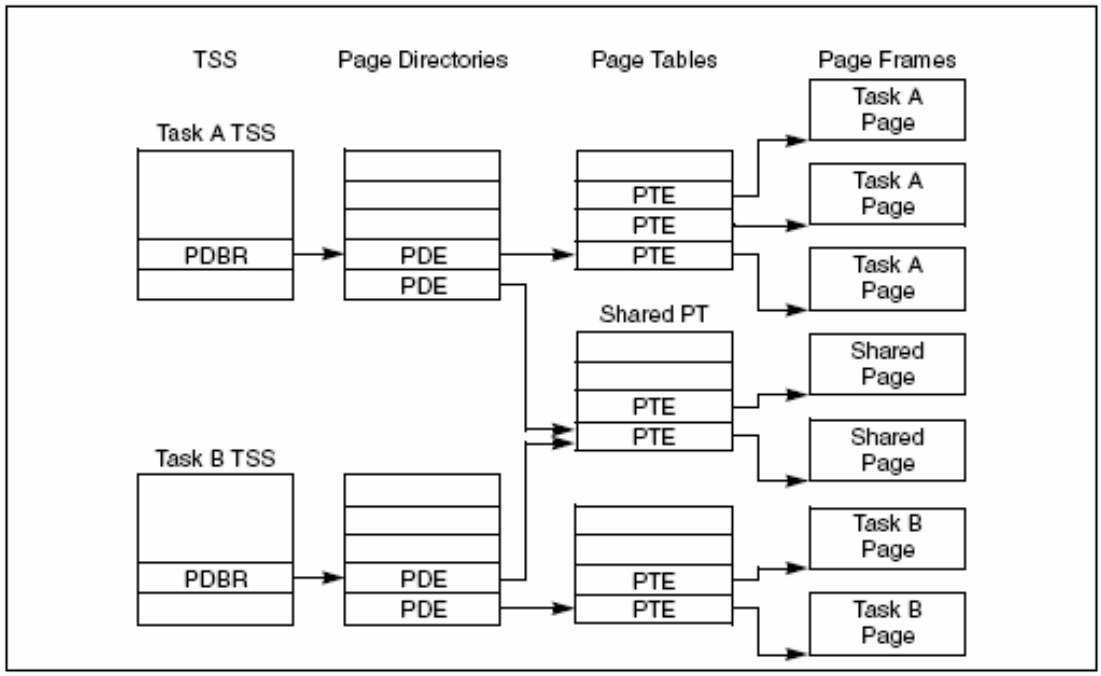

图 6-8 重叠线性——物理地址映射

6.5.2.任务逻辑地址空间

为了在任务间共享数据，使用下列任何一种技术都可建立针对数据段的共享逻辑——物理地址空间映射：

- 通过 GDT 中的段描述符。所有任务都必须具有访问 GDT 中段描述符的权限。如果 GDT 中有段描述符指向的线性地址空间中的段映射到共享物理地址空间（对所有任务而言）中，那么，所有任务都可共享这些数据和代码段。

- 通过一个共享的 LDT。两个或多个任务可以使用同样的 LDT，只要它们的 TSS 中的 LDT 域指向同一个 LDT。如果共享 LDT 中的某些段描述符指向的段被映射到物理地址空间中的同一个区域，则这些段中的数据和代码可在这些共享 LDT 的任务中共享。这种共享方法比通过 GDT 共享更具选择性，因为共享可以被限制到某些特定的任务上。系统中的其它任务可以有不同的 LDT，因为不给它们访问共享段的权限。
- 通过不同 LDT 中映射到同一个线性地址空间中的段描述符。如果对每个任务来说，这同一线性地址空间被映射到同一物理地址空间，则这些段描述符就允许任务共享这些段。这些段描述符通常被称为别名。这种共享方法比前面列出的两种更具选择性，因为 LDT 中的其它段描述符可以指向不共享的独立线性地址。

6.6.16 位任务状态段（TSS）

32 位 IA-32 处理器也能够像 Intel 286 那样识别 16 位 TSS 格式（参看 6-9）。它供那些为了与早期 IA-32 处理器兼容的软件使用的。

关于 16 位 TSS 有必要了解下列额外的信息：

- 不能使用 16 位 TSS 来实现一个虚拟 8086 任务。
- 16 位 TSS 的合法段界限是 2CH。
- 16 位 TSS 不包含页目录表的被加载到 CR3 控制寄存器基地址域。因而，16 位任务不支持每个任务拥有一个独立的页表集。如果 16 位任务被调度，则使用前一个任务的页表结构。
- 16 位 TSS 不包含 I/O 基地址，因此不支持 I/O 映射功能。
- 当任务状态被保存在 16 位 TSS 中时，EFLAGS 寄存器的高 16 位和 EIP 寄存器丢失。
- 当通用寄存器从 16 位 TSS 中被加载或者保存时，高 16 位被修改，但不维护。

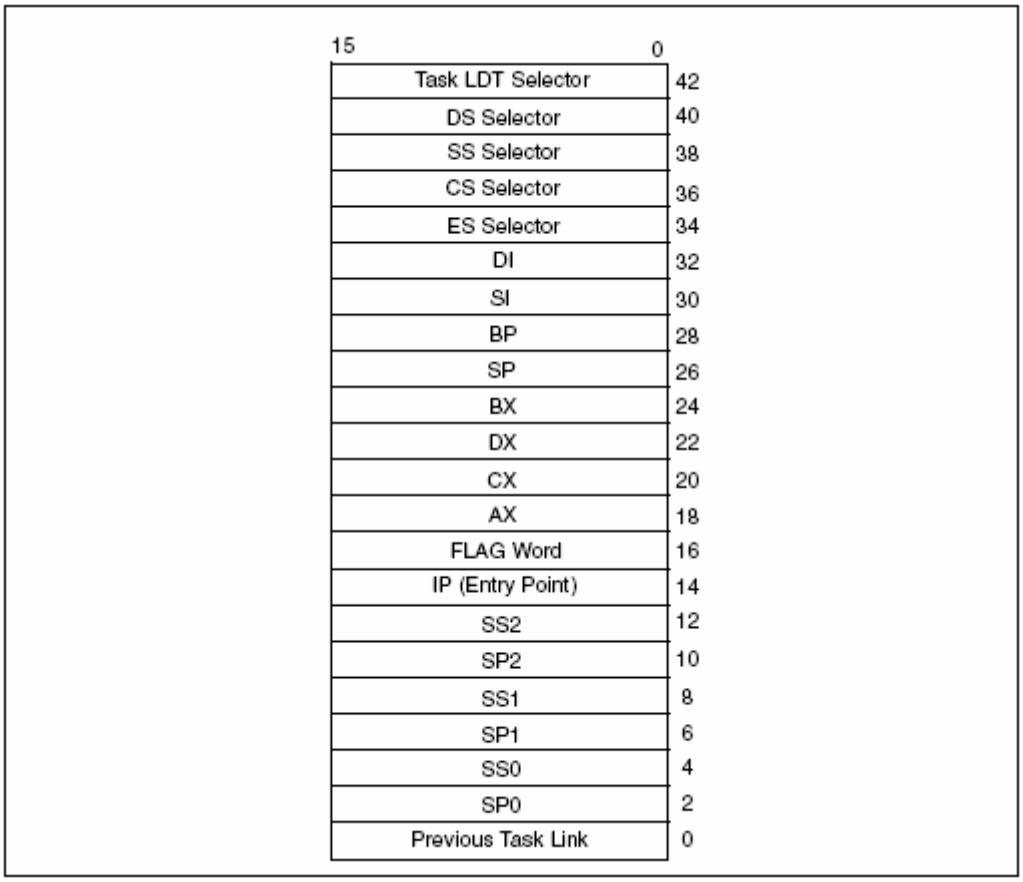

图 6-9 16 位 TSS 格式

第 7 章 多处理器管理

IA-32 架构有几种机制来管理和提升连接到同一条系统总线上的多个处理器的性能。这些机制包括：

- 总线加锁和/或高速缓存相干性（coherency）管理，以实现对系统内存的原子操作。
- 串行化指令。（这些指令仅用于 Pentium 4、Intel Xeon、P6 系列、Pentium 处理器。）
- 处理器芯片内置的高级可编程中断控制器（APIC）（参见第 8 章“高级可编程中断控制器（APIC）”）。APIC 架构是随着 Pentium 处理器被引入 IA-32 处理器的。
- 两级高速缓存（L2）。对于 Pentium 4、Intel Xeon、P6 系列处理器来说，L2 高速缓存包括在处理器的封装里，与处理器紧密相连。而对 Pentium 和 Intel486 处理器来说，则提供了用于支持外部 L2 高速缓存的引脚。
- 三级高速缓存（L3）。对于 Intel Xeon 处理器来说，L3 高速缓存包括在处理器的封装里，与处理器紧密相连。
- 超线程技术。这是对 IA-32 架构的扩展，能够让一个处理器内核并发地执行两个或两个以上的执行线索（参见 7.6.“超线程技术”）。

这些机制在对称多处理系统（SMP）中是极其有用的。但是，也可以用在共享系统总线的 IA-32 处理器和专用处理器（例如通信、图形、视频处理器）的应用程序中。

这些多处理机制的主要目的如下：

- 保持系统内存的相干性（coherency）——当两个或多个处理器试图同时访问系统内存的同一地址时，必须有某种通信机制或内存访问协议来提升数据的相干性，以及在某些情况下，允许一个处理器临时的锁定某个内存区域。
- 保持高速缓存的一致性（consistency）——当一个处理器访问另一个处理器高速缓存中的数据时，必须得到正确的数据。如果被访问的处理器修改了数据，那么，所有访问这个数据的处理器都要收到修改后的数据。
- 允许以可预知的顺序写内存——在某些情况下，从外部观察到的写内存的顺序

必须要和编程时指定的写内存顺序一致。

- 在一组处理器中分派中断处理——当几个处理器正在并行地工作在一个系统中时，有必要有一个集中的机制来接收中断并把它们派发到可用的处理器中请求服务。
- 采用现代操作系统和应用程序具有的多线程和多进程的特性来提升系统的性能。

IA-32 架构的高速缓存机制和缓存一致性在第 10 章“内存缓冲控制”中讨论，APIC 架构在第 8 章“高级可编程中断控制器（APIC）”中讨论，总线和内存加锁、串行化指令、内存排序、超线程技术在随后的几节中讨论。

7.1. 加锁的原子操作

32 位 IA-32 处理器支持对系统内存中的某个区域进行加锁的原子操作。这些操作常用来管理共享的数据结构（例如信号量、段描述符、系统段或者页表），两个或多个处理器可能会同时修改这些数据结构中的同一数据域或标志。处理器使用三个相互依赖的机制来实现加锁的原子操作：

- 保证原子操作。
- 总线加锁，使用 LOCK#信号和 LOCK 指令前缀。
- 高速缓存相干性协议，确保对高速缓存中的数据结构执行原子操作（高速缓存锁）。这种机制存在于 Pentium 4、Intel Xeon 和 P6 系列处理器中。

这些机制相互依赖的方式如下。某些基本的内存事务（例如在系统内存中读或写一个字节）要被保证为原子的。也就是说，一旦开始，处理器保证这个操作会在另一个处理器或总线代理访问相同的内存区域之前结束。处理器还支持总线加锁以实现所选的内存操作（例如在共享内存中的读——改——写操作），典型情况下，这些都必须是原子的操作，而不能以总线加锁的方式自动处理。因为频繁使用的内存区域经常被缓存在处理器的 L1、L2 高速缓存里，所以通常情况下处理器并不需要激活总线锁就可在它的内部实现原子的操作。这里，处理器的高速缓存相干性协议保证，在高速缓冲的内存区域中执行原子操作的同时，其它高速缓存了相同内存区域的处理器能得到正确的管理。

注意，这些处理加锁的原子操作的机制已经像 IA-32 处理器一样发展得越来越复

杂。最新的 IA-32 处理器（例如 Pentium 4、Intel Xeon 和 P6 系列处理器）提供了一种比早期 IA-32 处理器更为精简的机制。在下面的小节中我们会讨论这些。

7.1.1.保证原子操作

Pentium 4、Intel Xeon、P6 系列、Pentium 以及 Intel486 处理器保证下面的基本内存操作的执行总是原子的：

- 读或写一个字节。
- 读或写一个 16 位边界对齐的字。
- 读或写一个 32 位边界对齐的双字。

Pentium 4、Intel Xeon、P6 系列以及 Pentium 处理器还保证下列内存操作总是原子的：

- 读或写一个 64 位边界对齐的四字。
- 对与 32 位数据总线相匹配的未高速缓存的内存位置进行 16 位的访问。

P6 系列处理器还保证下列内存操作的执行总是原子的：

- 对与 32 字节高速缓冲线相匹配的高速缓存的内存进行非对齐的 16 位、32 位、64 位的访问。

对于 Pentium 4、Intel Xeon、P6 系列、Pentium 以及 Intel486 处理器来说，访问可被高速缓存的但是却被总线宽度、高速缓存线、页边界所分割的内存区域，不保证操作是原子的。Pentium 4、Intel Xeon、P6 系列等处理器提供总线控制信号，许可外部内存子系统来实现被分割访问的原子性。但是，非对齐的数据访问将严重影响处理器的性能，应该避免。

7.1.2.总线加锁

IA-32 处理器提供有一个 LOCK#信号，会在某些关键内存操作期间被自动激活，去锁定系统总线。当这个输出信号发出的时候，来自其它处理器或总线代理的总线控制请求将被阻塞。软件能够通过预先在指令前添加 LOCK 前缀来指定需要 LOCK 语义的其它场合。

在 Intel386、Intel486、Pentium 处理器中，明确地对指令加锁会导致 LOCK#信号的产生。由硬件设计人员来保证系统硬件中 LOCK#信号的可用性，以控制处理器间的内

存访问。

对于 Pentium 4、Intel Xeon 以及 P6 系列处理器，如果被访问的内存区域是在处理器内部进行高速缓存的，那么通常不发出 LOCK#信号；相反，加锁只应用于处理器的高速缓存（参见 7.1.4.LOCK 操作对处理器内部高速缓存的影响）。

7.1.2.1. 自动加锁

处理器自动遵循 LOCK 语义的操作如下：

执行引用内存的 XCHG 指令时。

- **设置 TSS 描述符的 B（忙）标志时。**在进行任务切换时，处理器测试并设置 TSS 描述符类型域中的忙标志。为了保证两个处理器不会同时切换到同一个任务，在测试和设置这个标志时处理器会遵循 LOCK 语义。
- **更新段描述符时。**在装入一个段描述符时，如果段描述符的访问标志被清除，则处理器会设置这个标志。在进行这个操作时，处理器会遵循 LOCK 语义。因此不会出现一个描述符正在被更新时，有其它处理器再来修改它的情况。为了保证这个操作有效进行，操作系统过程应该按照下列步骤来更新描述符：

  ——使用加锁的操作修改访问权限字节，以表明段描述符不存在，然后为类型域指定一个值，以表明描述符正在被更新。

  ——更新段描述符的域。（这个操作可能需要多次内存访问，因此不能使用加锁的操作。）

  ——使用一个加锁的操作来修改访问权限字节，以表明段描述符存在并且有效。

  注意，Intel386 处理器总是更新段描述符的访问标志，无论这个标志是否被清除。Pentium 4、Intel Xeon、P6 系列、Pentium 以及 Intel486 处理器仅在该标志被清除时才设置这个标志。
- **更新页目录表项和页表项时。**在更新页目录表项和页表项时，处理器使用加锁的周期来设置它们中的访问标志和脏标志。
- **响应中断。**发生中断请求后，中断控制器可能会使用数据总线给处理器传送中断向量。此时，处理器必须遵循 LOCK 语义来保证中断向量传送过程中数据总线上没有其它数据。

7.1.2.2.软件控制的总线加锁

为显式地强制执行 LOCK 语义，软件可以在下列指令修改内存区域时使用 LOCK 前缀。当 LOCK 前缀被置于其它指令之前或者指令没有对内存进行写操作（也就是说目标操作数在寄存器中）时，会产生一个非法操作码异常（#UD）。

- 位测试和修改指令（BTS、BTR、BTC）。
- 交换指令（XADD、CMPXCHG、CMPXCHG8B）。
- 自动假设有 LOCK 前缀的 XCHG 指令。
- 下列单操作数的算术和逻辑指令：INC、DEC、NOT、NEG。
- 下列双操作数的算术和逻辑指令：ADD、ADC、SUB、SBB、AND、OR、XOR。

一个加锁的指令会保证对目标操作数所在的内存区域加锁，但是系统可能会将锁定区域解释得稍大一些。

软件应该使用相同的地址和操作数长度来访问信号量（用作处理器之间发送信号的共享内存）。例如，如果一个处理器使用一个字来访问信号量，其它处理器就不应该使用一个字节来访问这个信号量。

总线锁的完整性不受内存区域对齐的影响。加锁语义会一直持续，以满足更新整个操作数所需的总线周期个数。但是，建议加锁访问应该对齐在它们的自然边界上，以提升系统性能：

- 任何 8 位访问的边界（加锁或不加锁）。
- 锁定的字访问的 16 位边界。
- 锁定的双字访问的 32 位边界。
- 锁定的四字访问的 64 位边界。

对所有其它的内存操作和所有可见的外部事件来说，加锁的操作都是原子的。只有取指令和页表操作能够越过加锁的指令。加锁的指令可用于同步一个处理器写数据而另一个处理器读数据的操作。

对 P6 系列处理器来说，加锁的操作会串行化所有未完成的加载和保存操作（也就是等待它们完成）。这条规则同样适用于 Pentium 4 和 Intel Xeon 处理器，但有一个例外：访问弱排序内存类型的加载操作可能不会被串行化。

加锁的指令不应该用来保证写的数据可以作为指令取回。

注意

当前版本的 Pentium 4、Intel Xeon、P6 系列、Pentium 和 Intel486 处理器的加锁的指令允许写的数据可以作为指令取回。但是 Intel 建议需要使用自修改代码的开发者使用另外一种同步机制。后面我们会讨论这个机制。

7.1.3. 处理自修改和交叉修改代码

处理器将数据写入当前执行代码段以实现将该数据作为代码来执行的目的的行为称为**自修改（self-modifying）代码**。IA-32 处理器在执行自修改代码时展现出的行为与特定模式相关，具体依赖于被修改的代码与当前执行位置之间的距离。由于处理器的体系结构变得越来越复杂，且在退役点（retirement point）之前就开始推测性地执行代码（如 Pentium 4、Intel Xeon、P6 系列等处理器中就是这样），因而，判断应该执行哪段代码——是修改前的还是修改后的——就变得模糊不清。要想写出自修改代码并确保它与现在和将来版本的 IA-32 架构兼容，必须选择下面的两种方式之一编写代码：

（方式 1）

将代码作为数据写入代码段；

跳转到新的代码位置或某个中间位置；

执行新的代码；

（方式 2）

将代码作为数据写入代码段；

执行一条串行化指令；（如：CPUID 指令）

执行新的代码；

（在 Pentium 或 Intel486 处理器上运行的程序不需要以上面的方式书写，但是为了与 Pentium 4、Intel Xeon、P6 系列处理器兼容，建议采用上面的两种方式之一。）

需要注意的是，自修改代码运行效率要低于非自修改代码或者正常代码，性能损失的程度依赖于修改的频率以及代码本身的特性。

一个处理器将数据写入另外一个处理器的代码段，以使得那个处理器将该数据作为代码执行的行为称为**交叉修改（cross-modifying）代码**。像自修改代码一样，IA-32 处理器执行交叉修改代码时展现出的行为与特定模式相关，具体依赖于被修改的代码与当前执行位置之间的距离。要想写出交叉修改代码并确保它与现在和将来版本的

IA-32 架构相兼容，必须实现下面的处理器同步算法。

；修改处理器的操作

Memory_Flag ← 0;（或者 1 之外的其它值）

将代码作为数据写入代码段;

Memory_Flag ← 1;

；执行处理器的操作

WHILE （Memory_Flag ≠ 1）

等待代码更新;

ELIHW;

执行串行化指令;（例如，CPUID 指令）

开始执行修改后的代码;

（在 Pentium 或 Intel486 处理器上运行的程序不需要以上面的方式书写，但是为了与 Pentium 4、Intel Xeon、P6 系列处理器兼容，建议采用上面的方式编写代码。）

像自修改代码一样，交叉修改代码运行效率低于非交叉修改（正常）代码，性能损失的程度依赖于修改的频率以及代码本身的特性。

7.1.4.加锁操作对处理器内部高速缓存的影响

对于 Intel486 和 Pentium 处理器来说，在进行加锁操作时，总是在总线上发出 LOCK#信号，即使锁定的内存区域已经高速缓存在处理器中。

对于 Pentium 4、Intel Xeon、P6 系列处理器而言，加锁操作期间，如果锁定的内存区域已经被高速缓存进正在执行回写内存加锁操作的处理器中，并且已经完全进入了高速缓冲线了，则处理器不对总线发出 LOCK#信号。相反，它将修改缓存区域中的数据，并依赖高速缓存相干性机制来保证加锁操作的执行是原子的。这个操作称为“高速缓存加锁”。高速缓存相干性机制会自动阻止两个或多个缓存了同一内存区域的处理器同时修改该区域的数据。

7.2.内存排序

术语**内存排序（memory ordering）**指的是处理器通过系统总线发出读（加载）和写（保存）系统内存的顺序。IA-32架构支持几种内存排序模型，具体依赖于架构的实

现。例如，Intel386处理器强制执行**编程排序**（又称为**强排序**），也就是说，在任何情况下，发送到系统总线上的读和写的顺序与它们在代码流中的顺序是一致的。

为了允许代码优化，在Pentium 4、Intel Xeon、P6系列处理器的IA-32架构中，允许不遵循**处理器排序**这种强排序模型。这些处理器排序的变体允许一些性能增强操作，诸如读可以放在缓冲的写前面。所有这些变体的目标都是，提高指令执行速度，同时要维持内存相干性，即使是在多处理器系统中也是如此。

接下来的部分分别介绍用于Intel486和Pentium处理器的内存排序模型以及用于Pentium 4、Intel Xeon、P6系列处理器内存排序模型。

7.2.1.Pentium 和 Intel486 处理器的内存排序

Pentium 和 Intel486 处理器遵循处理器排序的内存模型，但是，在大多数情况下，它们是强排序处理器。读写操作总是以编程时指定的顺序出现在系统总线上，除非正在进行处理器排序。当所有的缓冲的写操作都在缓存中命中，因此也就不会与未命中的读操作访问相同的内存地址时，未命中的读操作可以在总线上越过缓冲的写操作。

在执行 I/O 操作时，读操作和写操作总是以编程时指定的顺序执行。

想要在“处理器排序”的处理器（例如，Pentium 4、Intel Xeon、P6 系列处理器）上正确运行的软件不应该依赖 Pentium 或 Intel486 处理器的相对强排序。软件应该保证对共享变量的访问能够遵守编程顺序，这种编程顺序是通过使用加锁或串行化指令来完成的（参见 7.2.4.“强化和弱化内存排序模型”）。

7.2.2.Pentium 4、Intel Xeon、P6 系列处理器的内存排序

Pentium 4、Intel Xeon、P6 系列处理器也是使用“处理器排序”的内存模型，这种模型可以被进一步定义为“带有存储缓冲转发排序的写”。这个模型有下面的特点。

在一个单处理器系统中，对于定义为回写可缓冲的内存区域，下面的排序规则将被应用：

1．可以预先地和以任意顺序执行读。

2．读可以越过缓冲的写，但处理器是自我统一的。

3．对内存的写操作总是以编程顺序执行，除了执行 CLFUSH 指令的写和利用非瞬时的移动指令（MOVNTI、MOVNTQ、MOVNTDQ、MOVNTPS、MOVNTPD）来执行的流存储（写）。

4．写可以被缓冲。

5．写不能够预先执行；它们只能等到其它指令执行完毕。

6．来自缓冲写的数据可以直接被转发到处理器内等待中的读。

7．读或写不能跨越 I/O 指令、加锁的指令或者串行化指令。

8．读不能越过 LFENCE 和 MFENCE 指令。

9．写不能越过 SFECE 和 MFENCE 指令。

第二条规则允许读越过写。然而如果写和读都是访问同一个内存区域，那么处理器内部的监测机制将会检测到冲突并且在处理器使用错误的数据执行指令之前更新已经缓存的读。

第六条规则构成了一个对其它写排序模型的例外。

注意，“带有存储缓冲转发的排序的写操作”（本节开始的地方介绍的）指的是第 2 条规则和第 6 条规则组合之后产生的效果。

一个多处理器系统遵循下面的排序规则：

- 每个处理器使用在单个处理器系统中同样的排序规则。
- 所有处理器所观察到的某个处理器的写操作的顺序是相同的。
- 来自单个处理器的写操作在系统总线上的排序并不与其它处理器的相关。

图 7-1 的例子可以说明后一条规则。假设，在一个三处理器的系统中，每个处理器执行三个写操作，分别对着 A、B、C 这三个地址。每个处理器以编程的顺序执行写操作，但是由于总线仲裁和其它的内存访问机制，三个处理器执行写操作的顺序可能每次都不相同。地址 A、B、C 处的最终值会因这个写操作序列的每次执行的不同而改变。

本节介绍的处理器排序模型与 Pentium 和 Intel486 处理器使用的模型是一样的。在 Pentium 4、Intel Xeon、P6 系列处理器中唯一强化的地方是：

- 增加了对推测性读操作的支持。
- 存储缓冲转发，当一个读操作越过一个访问相同地址的写操作时。
- 来自长串存储和串移动操作的无次序保存（参见后面 7.2.3.“Pentium 4、Intel Xeon、P6 系列处理器的串操作的无次序存储”）。

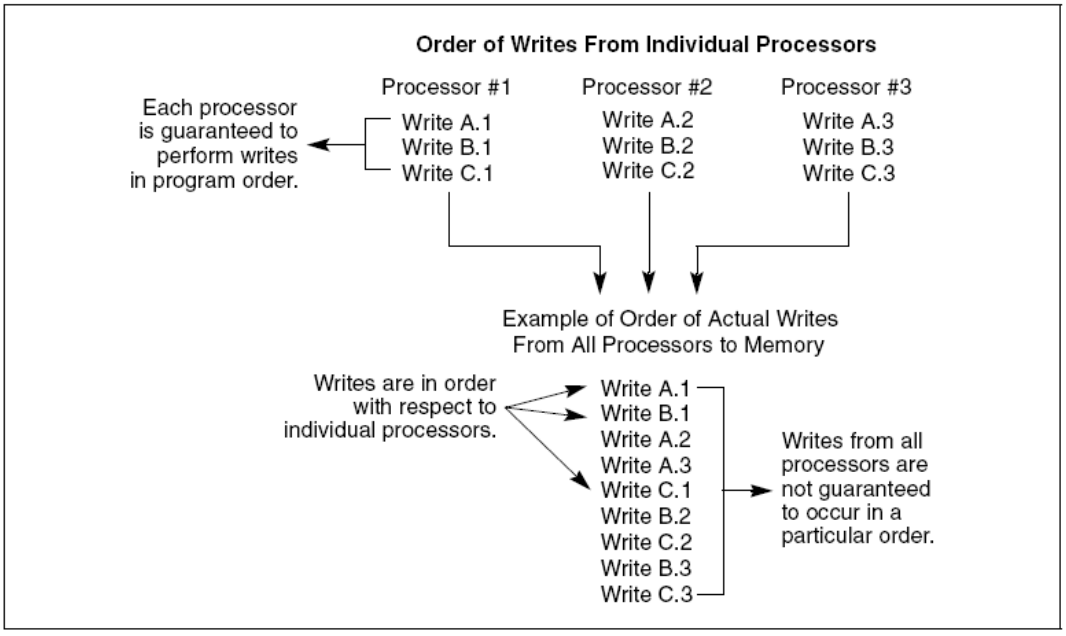

Figure 7-1. Example of Write Ordering in Multiple-Processor Systems

图 7-1 多处理器系统中写串行化的例子

7.2.3.Pentium 4、Intel Xeon、P6 系列处理器的串操作的无次序存储

在串存储的操作期（MOVS 和 STOS 指令引起的）间，Pentium 4、Intel Xeon、P6 系列处理器修改处理器的操作来最大化性能。一旦“快速串”的条件满足了（将在下面介绍），处理器实际上会在缓冲线上以缓冲线模式进行串操作（从外面看）。这会导致处理器在循环过程中发出对源地址的缓冲线读请求，以及在外部总线上发出对目标地址的写请求，并且已知了目标地址内的数据串一定要被修改。在这种模式下，处理器仅仅在缓冲线边界时才会接受中断。因此，目标数据的失效和存储可能会以不规则的顺序出现在外部总线上。

依赖于顺序存储排序的代码不应该对将要存储的整个数据结构使用串操作指令。数据和信号量应该分开。依赖于顺序的代码应该在每一次串操作之后，使用唯一的一个已保存的分离的信号量，来允许正确排序的数据被所有处理器看见。

“快速串”操作的初始条件是：

- 在 Pentium III 处理器中，EDI 和 ESI 必须是 8 位对齐的。在 Pentium 4 中，EDI 必须是 8 位对齐的。

- 串操作必须是按地址增加的方向进行的。
- 初始操作计数器（ECX）必须大于等于 64。
- 源和目的内存的重叠区域一定不能小于一个缓冲线的大小(Pentium 4 和 Intel Xeon 处理器中是 64 个字节；P6 系列和 Pentium 处理器是 32 个字节）。
- 源地址和目的地址的内存类型必须是 WB 或 WC。

7.2.4.强化或弱化内存排序模型

IA-32 架构提供了几种机制用来强化或弱化内存排序模型，以处理特殊的编程情形。这些机制包括：

- I/O 指令、加锁指令、LOCK 前缀以及串行化指令等，强制在处理器上进行较强的排序。
- SFENCE 指令（在 Pentium III 中引入）和 LFENCE、MFENCE 指令（在 Pentium 4 和 Intel Xeon 处理器中引入）提供了某些特殊类型内存操作的排序和串行化功能。
- 内存类型范围寄存器（MTRR）可以被用来强化和弱化物理内存中特定区域的内存排序模型。（参看 10.11.“内存类型范围寄存器（MTRR）”）。MTRR 只存在于 Pentium 4、Intel Xeon、P6 系列处理器中。
- 页属性表可以被用来强化某个特殊页或一组页的内存排序（参看 10.12.“页属性表”（PAT））。PAT 只存在于 Pentium 4、Intel Xeon、P6 系列处理器中。

这些机制可以通过下面的方式使用。

总线上的内存映射设备和其它 I/O 设备通常对向它们缓冲区写操作的顺序很敏感。I/O 指令（IN 和 OUT 指令）以下面的方式对这种访问执行强写操作的排序。在执行一条 I/O 指令之前，处理器等待之前的所有指令执行完毕以及所有的缓冲区都被写入了内存。只有取指令和页表查询（page table walk）能够越过 I/O 指令。后续指令要等到 I/O 指令执行完毕才开始执行。

多处理器系统中的同步机制可能会依赖于“强排序”模型。这里，程序可能会使用诸如 XCHG 或者带 LOCK 前缀的加锁指令，来保证对内存的读——改——写操作是原子的。典型情况下，加锁操作就像 I/O 指令一样，因为它们都要等待所有之前的指令

执行完毕以及缓冲的写都被写入了内存（参看 7.1.2.“总线加锁”）。

程序同步可以通过串行化指令来实现（参看 7.4.“串行化指令”）。这些指令通常用于临界过程或者任务边界来保证之前所有的指令在跳转到新的代码区或场境切换之前执行完毕。像 I/O 和加锁指令一样，处理器等待之前所有的指令执行完毕以及所有的缓冲的写保存到内存后才开始执行串行化指令。

SFENCE、LFENCE 和 MFENCE 指令提供了一种高效的方式，来保证在产生弱排序结果的程序和读取这个数据的程序之间进行读写内存的排序。这些指令的功能如下：

- SFENCE——串行化程序指令流中发生在 SFENCE 指令之前的所有存储（写）操作，但是不影响加载操作。
- LFENCE——串行化程序指令流中发生在 SFENCE 指令之前的所有加载（读）操作，但是不影响存储操作。
- MFENCE——串行化程序指令流中发生在 MFENCE 指令之前的所有存储和加载操作。

注意，SFENCE、LFENCE、MFENCE 指令提供了一个比 CPUID 指令更加有效的控制内存排序的方法。

MTRR 在 P6 系列处理器中引入，用来定义物理内存的特定区域的高速缓存特性。下面的两个例子是关于如何利用 MTRR 设置的内存类型来强化和弱化 Pentium 4、Intel Xeon、P6 系列处理器的内存排序：

- 强非缓冲（UC）的内存类型强制对内存访问实行强排序模型。这里，所有对 UC 内存区域的读写都出现在总线上，并且次序混乱的或推测性的访问都不被执行。这种内存类型可以应用于映射成 I/O 设备的内存区域，来强制执行强内存排序。
- 对于可以容忍弱排序的内存区域来说，可以选择回写（WB）内存类型。这里，读可被推测性地执行，写可被缓冲和组合。对于这种类型的内存，高速缓存加锁是通过一个加锁的原子操作实现的，这个操作不会分割缓存线，因此会减少典型的同步指令（如，XCHG 在整个读——改——写操作周期要锁定总线）所带来的性能损失。对于 WB 内存，如果内存访问是在高速缓存线内进行的，则 XCHG 指令会锁定高速缓存而不是数据总线。

PAT 是在 Pentium III 中引入的，用来增强高速缓存的性能，即可被指定给数个页或者数个页组。PAT 机制通常被用来加强页级别的高速缓存性能，即 MTRR 建立的高速

缓冲特性。表 10-7 显示了 PAT 与 MTRR 的相互作用。

建议，为运行在 Pentium 4、Intel Xeon、P6 系列处理器上而写的软件，要假定是在处理器排序模型或者较弱的内存排序模型之下。Pentium 4、Intel Xeon、P6 系列处理器没有实现强内存排序模型，除非使用 UC 内存类型。尽管 Pentium 4、Intel Xeon、P6 系列处理器支持处理器排序模型，但是，Intel 并没有保证将来的处理器会支持这种模型。为了使软件可移植到将来的处理器，操作系统最好提供临界区和资源控制构想，以及基于 I/O、加锁、串行化指令的 API，用于同步多处理器系统对共享内存区的访问。同时，软件不应该依赖于系统硬件不支持内存模型的处理器排序。

7.3.向多个处理器传播页表项和页目录表项的修改

在多处理器系统中，当一个处理器修改一个页表项或页目录表项时，这个修改必须要传播到其余所有的处理器。这个过程通常称为“TLB 击毁（shootdown）”。修改页表项或页目录表项的传播可以用基于内存的信号量或者处理器间中断（IPI）来完成。例如，IA-32 架构上的一个简单的算法正确的 TLB 击毁的序列步骤如下：

1．开始设置屏障——停止除一个处理器之外的所有其它处理器；也就是，让除了一个处理器之外的所有其它处理器执行停机指令或者停在一个自旋循环上。

2．让那个活动着的处理器修改 PTE 或 PDE。

3．让所有处理器在它们各自的 TLB 中使修改的 PTE、PDE 失效。

4．结束屏障——恢复所有的处理器；恢复一般的处理。

也可以开发替换的、性能优化的 TLB 击毁算法；但是，开发者一定要保证满足下面的两个条件之一：

- 在更新过程中，不同的处理器上不使用不同的 TLB 映射。
- 操作系统要处理好更新过程中处理器使用旧的映射的情况。

7.4.串行化指令

IA-32 架构定义了几个串行化指令。这些指令强制处理器完成先前指令对标志、寄存器以及内存的修改，并且在下一条指令取出和执行之前将所有缓冲的写保存到内存。例如：当用 MOV 指令将一个操作数装入 CR0 寄存器以开启保护模式时，处理器必须在进入保护模式之前执行一个串行化操作。这个串行化操作保证所有在实地址模式下开

始执行的指令在切换到保护模式之前都执行完毕。

串行化指令的概念在 Pentium 处理器中被引入 IA-32 架构的。这种指令对于 Intel486 或更早的处理器是没有意义的，因为它们并没有实现并行指令执行。

非常值得注意的是，在 Pentium 4、Intel Xeon、P6 系列处理器上执行串行化指令会限制推测性执行，因为推测性执行的指令的结果被丢弃了。

下面的指令是串行化指令：

- 特权串行化指令——MOV（目标操作数为控制寄存器）、MOV（目标操作数为调试存器）、WRMSR、INVD、INVLPG、WBINVD、LGDT、LLDT、LIDT、LTR。
- 非特权串行化指令——CPUID、IRET、RSM。
- 非特权内存排序指令——SFENCE、LFENCE、MFENCE。

当处理器串行化指令的执行时，它保证在执行下一条指令之前，所有待处理的内存事务都要完成，包括保存在它的存储缓冲区中的写。禁止任何指令越过串行化指令，串行化指令也不能越过其它指令（读、写、取指令或者 I/O）。

可以在任何特权级下执行 CPUID 指令，来串行化指令执行而不影响程序的执行流，EAX、EBX、ECX、EDX 寄存器的修改除外。

SFENCE、LFENCE、MFENCE 指令提供了更多的粒度，用以控制内存加载和存储的串行化（参见 7.2.4.“强化和弱化内存排序模型”）。

关于串行化指令还须注意下面的附加信息：

- 当它串行化指令执行的时候，处理器并不将它的数据高速缓存中的被修改的数据内容回写到外部内存中。软件可以通过执行 WBINVD 指令强制把修改的数据写回，WBINVD 是一个串行化指令。必须注意的是，频繁使用 WBINVD 指令会严重降低系统的性能。
- 当执行一条指令来开启或者关闭分页（也就是改变控制寄存器 CR0 的 PG 标志）时，这条指令后面应该跟一条跳转指令。跳转目标应该是在 PG 标志的新设置（开启或关闭分页）下被取出的，但跳转指令本身还是在先前的设置下被取出的。Pentium 4、Intel Xeon、P6 系列处理器不需要在设置 CR0 处理器之后放置跳转指令（因为任何对 CR0 进行操作的 MOV 指令都是串行化的）。但是为了与其它 IA-32 处理器向前和向后兼容，最好是放置一条跳转指令。
- 无论何时，执行一条指令改变 CR3 的内容开启分页时，则下一条指令会根据 CR3 新内容所对应的转换表进行取指令操作。因此下一条以及之后的指令应该

根据新的 CR3 内容建立进行映射。（TLB 中的全局项不失效，参见 10.9.“使转换后备缓冲区（TLB）失效”。）

- Pentium 4、Intel Xeon、P6 系列以及 Pentium 处理器使用分支预测技术来提升系统性能——通过在分支指令执行之前预取分支的目标指令。因此，在执行分支指令时，指令不是串行化的。

7.5.多处理器（MP）初始化

IA-32 架构（从 P6 系列处理器开始）定义了一个多处理器（MP）初始化协议，称为“多处理器规范 1.4 版”（Multiprocessor Specification Version 1.4）。这个规范定义了多处理器系统中的 IA-32 处理器使用的引导协议（这里，多处理器定义为两个或两个以上的处理器）。MP 初始化协议具有下列重要特征：

- 它支持多处理器有控制的自举，无需专门的系统硬件。
- 它允许硬件启动系统的自举，无需专门的信号或者预先定义好的引导处理器。
- 允许所有的 IA-32 处理器以相同的方式自举，包括那些支持超线程技术的处理器。

执行 MP 初始化协议的机制因处理器型号而异，具体如下：

- P6 系列的处理器——BSP 和 AP（参见 7.5.1.“BSP 和 AP 处理器”）的选择是通过 APIC 总线上的仲裁来处理的，使用 BIPI 和 FIPI 消息。参见附录 C，那里完整地讨论了 P6 系列处理器的 MP 初始化。
- Intel Xeon 处理器，其系列、模型和分级 ID 在 F09H 以内的——BSP 和 AP（参见 7.5.1.“BSP 和 AP 处理器”）的选择是通过系统总线上的裁决来处理的，使用 BIPI 和 FIPI 消息。参看 7.5.3.“Intel Xeon 处理器的 MP 初始化协议算法”，那里有完整的讨论。
- Intel Xeon 处理器，其系列、模型和分级 ID 等于或高于 F09H 的——BSP 和 AP 的选择是通过特殊的系统总线周期来处理的，不需要使用 BIPI 和 FIPI 消息来进行裁决。这种选择方法在 7.5.3.“Intel Xeon 处理器的 MP 初始化协议算法”中也有讨论。

处理器的系列、模型和分级 ID 可以通过执行 CPUID 指令得到——以 EAX 为 1 执行这条指令后，结果就会保存到 EAX 寄存器中。

7.5.1. BSP 和 AP 处理器

MP 初始化协议定义了两类处理器：自举处理器（BSP）和应用处理器（AP）。在加电或复位之后，系统硬件会动态的选择一个处理器作为 BSP，余下的处理器被指派为 AP。

作为 BSP 选择机制的一部分，BSP 的 IA32_APIC_BASE MSR（参看图 8-5）中的 BSP 标志要置位，以表明这个处理器是一个 BSP。其余所有的处理器要清除这个标志。

BSP 执行 BIOS 的自举代码来配置 APIC 环境，建立起系统范围内的数据结构，启动和初始化 AP。当 BSP 和 AP 被初始化后，BSP 就开始执行操作系统的初始化代码。

在加电或复位之后，AP 完成一个最小的自我配置，然后等待来自 BSP 的启动信号（SIPI 消息）。一旦收到 SIPI 消息的时候，AP 执行 BIOS 的 AP 配置代码，结束后把 AP 置于停机状态。

在支持超线程技术的 IA-32 处理器中，MP 初始化协议将系统总线上的每个逻辑处理器都看作是一个独立的处理器（带有唯一的 APIC ID）。在启动的时候，一个逻辑处理器被选为 BSP，其余的被指派为 AP。

7.5.2. Intel Xeon 处理器的 MP 初始化协议的需求和限制

MP 初始化协议对系统有下列的需求和限制：

- MP 协议仅在加电或复位之后才执行。如果 MP 协议已经完成，并且一个 BSP 也被选定，则后面的 INIT（不管是对某个特定的处理器还是系统范围内的所有处理器发出的）不会导致 MP 协议的再次执行。相反，每一个处理器都会检查自己的 BSP 标志（在 IA32_APIC_BASE MSR 中）来决定是否需要执行 BIOS 自举代码（如果它是 BSP）或者进入等待 SIPI 的状态（如果它是 AP）。
- 系统中所有的可以发出中断的设备都必须在 MP 初始化时候被禁止。禁止中断的时间包括从 BSP 对 AP 发出 INIT——SIPI——SIPI 序列时到 AP 对最后一个 SIPI 响应时的这个时间窗。

7.5.3. Intel Xeon 处理器的 MP 初始化协议算法

在加电或复位之后，系统中的 Intel Xeon 处理器执行 MP 初始化协议算法来初始

化系统总线上的所有处理器。在执行这个算法的时候，将执行下面的启动和初始化操作：

1.系统总线上的每个处理器都被赋予一个唯一的基于系统拓扑结构的 8 位 APIC ID（参看 7.5.5.“在 MP 系统中识别处理器”）。这个 ID 被写入每个处理器的本地 APIC ID 寄存器中。

2．每个处理器都被分配一个唯一的基于它的 APIC ID 的仲裁优先级。

3．总线上的所有处理器都同时执行内部的 BIST。

4．完成 BIST 后，处理器使用硬件定义的选择机制从总线上可用的处理器中选择 BSP 和 AP。BSP 选择机制因处理器的系列、模型和分级 ID 的不同而不同：

——系列、型号和分级ID等于或大于F0AH的：

- 处理器开始监听 BNR#信号，这是个反复电路。当 BSR#引脚停止切换（反复）时，每个处理器都尝试在总线上发送一个 NOP 特殊周期。
- 具有最高仲裁优先级的处理器成功发出 NOP 周期，并被选为 BSP。该处理器设置自己 IA32_APIC_BASE MSR 中的 BSP 标志，然后从复位向量（物理地址是 FFFF FFF0H）所指的地方开始取出并执行 BIOS 自举代码。
- 余下的处理器（没有成功发出 NOP 特殊周期的）被指派为 AP。它们保持它们的 BSP 标志的清除状态并进入等待 SIPI 状态。

——系列、型号和分级ID低于F0AH的：

- 每一个处理器都对所有的处理器（包括自己）广播一个 BIPI。第一个广播 BIPI（因此收到了它自己发送的 BIPI 向量）的处理器，选择自己为 BSP 并在自己的 IA32_APIC_BASE MSR 中设置 BSP 标志。（参看 C.1.“P6 系列处理器的 MP 初始化过程概述”，那里描述了 BIPI、FIPI、SIPI 消息。）
- 余下的处理器（没有被选为 BSP 的）被指派为 AP。它们保持它们的 BSP 标志的清除状态并进入等待 SIPI 状态。
- 刚确立的 BSP 对所有的处理器（包括自己）广播一个 FIPI 消息，这是 MP 初始化的结束信号。只有设置了 BSP 标志的处理器才能响应这个 FIPI 消息，它从复位向量（物理地址 FFFF FFF0H）所指的地方开始取出并执行 BIOS 自举代码。

5．作为自举代码的一部分，BSP 建立一个 ACPI 表和一个 MP 表，并将自己的 APIC ID 加入到这些表中。

6．自举过程结束时，BSP 设置一个处理器计数器为 1，然后对系统中所有的 AP 广

播一个 SIPI 消息。这里，SIPI 消息包含一个指向 BIOS AP 初始化代码的向量（在 000VV000H 处，这里，VV 表示包含在 SIPI 中的向量）。

7．AP 初始化的第一步是建立起一个对 BIOS 初始化信号量的竞争（在 AP 之间）。第一个取得该信号量的 AP 开始执行初始化代码（参看 7.5.4.“MP 初始化举例”中关于信号量实现的细节）。作为 AP 初始化的一部分，AP 将它的 APIC ID 加入到 ACPI 和 MP 表中，并将处理器计数器加 1。初始化过程结束时，AP 执行 CLI 指令并进入停机状态。

8．当每个 AP 获得了信号量并执行了 AP 初始化代码后，BSP 就创立一个连接到系统总线上的处理器个数计数，完成 BIOS 自举代码的执行，之后开始执行操作系统的自举和启动代码。

9．在 BSP 执行操作系统的自举和启动代码时，AP 保持停机状态。此时，它们只能响应 INIT、NMI 和 SMI。也可以响应探听和 STPCLK#引脚的激活。

下节将给出一个运行在一个 MP 配置下的多 Intel Xeon 处理器中的 MP 初始化协议的例子（有代码）。

附录 B“模式相关寄存器（MSR）”将描述 MP 初始化完成之后，怎样对处理器的本地 APIC 的 LINT[0:1]引脚进行编程。

7.5.4.MP 初始化举例

下面的例子演示了在 BSP 和 AP 已经确立后，如何利用 MP 初始化协议来初始化 MP 系统中的 IA-32 处理器。这段代码运行在使用 MP 初始化协议的 IA-32 处理器中，包括 P6 系列处理器、Pentium 4 处理器、Intel Xeon 处理器（有或没有超线程技术）。

下面的常量和数据定义将在代码举例中使用。它们是基于表 8-1 中定义的 APIC 寄存器地址的。

```
ICR_LOW        EQU 0FEE00300H
SVR            EQU 0FEE000F0H
APIC_ID        EQU 0FEE00020H
LVT3           EQU 0FEE00370H
APIC_ENABLED   EQU 0100H
BOOT_ID        DD ?
```

```
COUNT          EQU 00H
VACANT         EQU 00H
```

7.5.4.1.典型的BSP初始化顺序

BSP 和 AP 选定以后（通过硬件协议，参看 7.5.3.“Intel Xeon 处理器的 MP 初始化协议算法”），BSP 开始执行从起始地址 FFFF FFF0H 开始的 BIOS 自举代码（POST）。自举代码通常完成下列操作：

1．初始化内存。

2．加载微代码更新到处理器中。

3．初始化 MTRR。

4．开启高速缓存。

5．以 EAX 为 0H 执行 CPUID 指令，然后读取 EBX、ECX、EDX 寄存器来决定 BSP 是不是一个“GenuineIntel”。

6．以 EAX 为 1H 执行 CPUID 指令，然后将 EAX、ECX、EDX 寄存器的值保存在系统内存的配置空间中，供以后使用。

7．装入 AP 的启动代码，放入位于 1M 内存低地址处的一个 4K 字节的页中。

8．切换到保护模式并确保 APIC 地址空间被映射到强非缓存（UC）的内存类型中。

9．从本地的 APIC ID 寄存器（默认为 0）读取 BSP 的 APIC ID：

```
MOV ESI，APIC_ID       ；本地APIC ID寄存器的地址
MOV EAX，[ESI]
AND EAX，0FF000000H    ；除了APIC ID外，其它位都清为零
MOV BOOT_ID，EAX       ；保存在内存中
```

然后将 APIC ID 保存到 ACPI 和 MP 表以及系统内存的配置空间中。

10．将 AP 启动代码的 4K 字节页的基地址转换为 8 位的向量。8 位在实地址模式的地址空间（1M 字节空间）中定义了 4K 字节页的地址。例如，0BDH 向量表示启动代码的地址为 000BD000H。

11．通过设置 APIC 虚假向量寄存器（SVR）的第 8 位来开启本地 APIC。

```
MOV ESI，SVR                ；SVR的地址
MOV EAX，[ESI]
OR EAX，APIC_ENABLED        ；设置第8位开启（0是复位）
```

```
MOV [ESI], EAX
```

12. 通过给 APIC 错误处理器设置一个 8-位向量来建立 LVT 错误处理入口。

```
MOV ESI, LVT3
MOV EAX, [ESI]
AND EAX, FFFFFF00H    ; 清除以前的向量
OR EAX, 000000xxH     ; xx是APIC错误处理程序的8位向量
MOV [ESI], EAX
```

13. 初始化加锁信号量 VACANT 为 00H。AP 使用这个信号量来决定执行 BIOS AP 初始化代码的顺序。

14. 执行下列操作设置 BSP 来探测系统中 AP 的存在以及 AP 的个数：

——将COUNT的值设为1。

——启动一个计时器（大概是间隔100毫秒设置一次）。在AP BIOS初始化代码中，AP将会增加COUNT变量。当计时器到期了，BSP如果计时器到期了而COUNT没有增加，表明没有AP或者发生了错误。

15.广播一个 INIT——SIPI——SIPI 的 IPI 序列到 AP，来唤醒它们并进行初始化：

```
MOV ESI、ICR_LOW ; load address of ICR low dword into ESI
MOV EAX、000C4500H ; load ICR encoding for broadcast INIT IPI
                   ; to all APs into EAX
MOV [ESI]、EAX  ; broadcast INIT IPI to all APs
                   ; 10-millisecond delay loop
MOV EAX、000C46XXH ; load ICR encoding for broadcast SIPI IP
                   ; to all APs into EAX、where xx is the
                   ; vector computed in step 10.
MOV [ESI]、EAX  ; broadcast SIPI IPI to all APs
                   ; 200-microsecond delay loop
MOV [ESI]、EAX  ; broadcast second SIPI IPI to all APs
; 200-microsecond delay loop
Step 15:
MOV EAX、000C46XXH ; load ICR encoding from broadcast SIPI IP
; to all APs into EAX where xx is the vector computed in step 8
```

16．等待计时器中断。

17．读取和评估 COUNT 变量，建立处理器计数。

18．如果有必要，重新配置 APIC 并继续执行余下的系统诊测。

7.5.4.2.典型的AP初始化顺序

当 AP 收到 SIPI 时，开始执行 BIOS AP 初始化代码，开始地址在 SIPI 中编码。AP 初始化代码一般执行下面的操作：

1．等待 BIOS 初始化信号量。获得信号量后，开始初始化。

2．装入微码更新。

3．初始化 MTRR（利用 BSP 使用的同样的映射）。

4．开启高速缓存。

5．以 EAX 寄存器为 OH 执行 CPUID 指令，然后读取 EBX、ECX、EDX 寄存器来决定 AP 是一个“GenuineIntel”。

6．以 EAX 为 1H 执行 CPUID 指令，然后将 EAX、ECX、EDX 寄存器的值保存在系统内存的配置空间中，供以后使用。

7．切换到保护模式并确保 APIC 地址空间被映射成强不可缓冲（strong unchaeable、UC）类型。

8．从本地的 APIC ID 寄存器读取 AP 的 APIC ID，将其加入到 MP 和 ACPI 表中，以及系统配置空间中。

9.通过设置 SVR 寄存器的第 8 个位以及建立用于错误处理的 LVT3(错误 LVT,error LVT）来初始化并配置本地 APIC（如 7.5.4.1“典型的 BSP 初始化顺序”的第 9 步和第 10 步）。

10．配置 AP 的 SMI 执行环境。（每个 AP 和 BSP 必须有一个不同的 SMBASE 地址）。

11．COUNT 加 1。

12．释放信号量。

13．执行 CLI 和 HLT 指令。

14．等待一个 INIT 这样的 IPI。

7.5.5.在 MP 系统中识别处理器

在 BIOS 完成了 MP 初始化协议后，每个处理器都可以通过它们的本地 APIC ID 被识别。软件可以通过下面的方式来访问这些 APIC ID：

- **读取本地 APIC 的 APIC ID。**处理器上运行的代码可以通过 MOV 指令来读取本地 APIC ID 寄存器的值（参看 8.4.6.“本地 APIC ID”）。
- **读取 ACPI 或者 MP 表。**作为 MP 初始化的一部分，BIOS 建立一个 ACPI 表和一个 MP 表。这些表在多处理器规范 1.4 版中定义。表中提供了系统中的处理器列表以及它们的 APIC ID。ACPI 表的格式来源于 ACPI 规范，这是一个用于 MP 系统的电源管理和平台配置规范的工业标准。

对于 Intel Xeon 处理器，在启动和初始化时分配的 APIC ID 是 8 位（参看图 7-2）。这里，第 1、2 位构成了两位处理器标识（也可以被认为是一个套接字标识）。在将处理器配置为集群的系统中，第 3 位和第 4 位构成两位的集群 ID。Intel Xeon 处理器的 MP 中的第 0 位用来标识同一个组合包中的两个逻辑处理器（参看 7.7.5.“识别 MP 系统中的逻辑处理器”）。对于不支持超线程技术的 Intel Xeon 处理器，第 0 位总是设为 0；对于支持超线程技术的 Intel Xeon 处理器，第 0 位的功能同 Intel Xeon 处理器的 MP 中的一样。

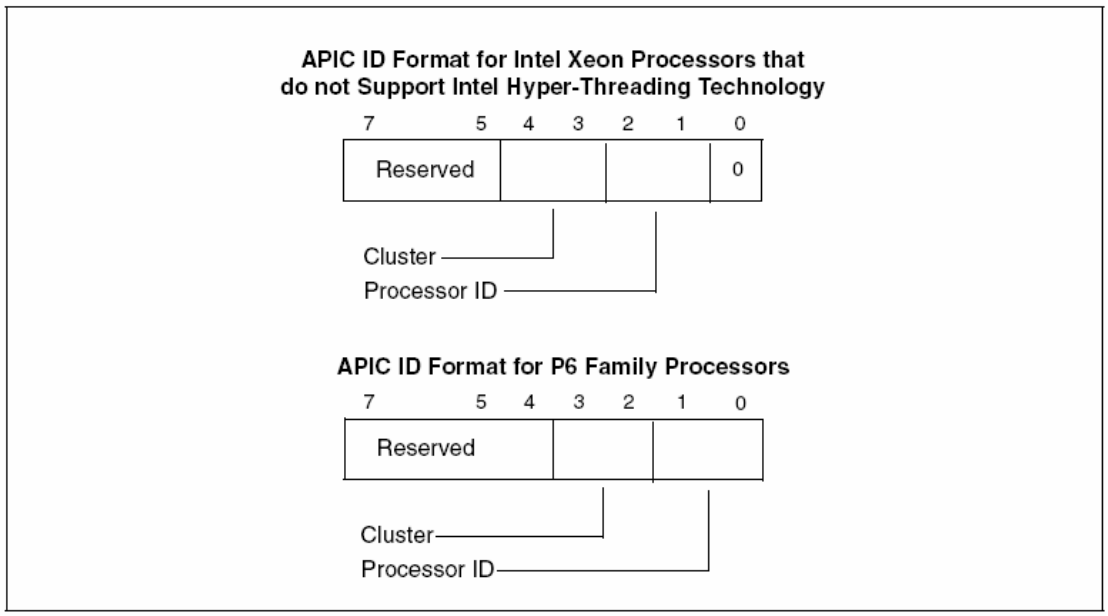

Figure 7-2. Interpretation of APIC ID in MP Systems

对于 P6 系列处理器，在启动和初始化时分配的 APIC ID 是 4 位（参看图 7-2）。这里，第 0、1 位构成两位的处理器（或套接字）标识。第 2、3 位构成两位集群 ID。

7.6.超线程技术

超线程技术是在 Intel Xeon 处理器的 MP 及 Intel Xeon 处理器的后期分级中被引入 IA-32 架构。所有支持超线程技术的 Pentium 4 处理器都支持这个技术。所有的超线程技术的配置都需要一个芯片组、BIOS 的支持，同时操作系统也必须针对超线程进行优化。参看 www.intel.com/info/hyperthreading 和《第 1 卷:基础架构》中的 2.2.4.“超线程技术”，获取更多有关信息。

Intel 建议，软件不要依赖 IA-32 处理器的名字来判断是否支持超线程技术，而应该使用 7.6.3.“探测超线程技术”中介绍的 CPUID 指令来确定。

超线程技术是对 IA-32 架构的一个扩展，它使一个物理处理器能同时执行两个或更多代码流（称作线程）。下面讨论 IA-32 处理器如何实现这个特征。

7.6.1.Intel 的超线程技术架构

图 7-3 显示了支持超线程技术的 IA-32 处理器的大体结构，这里以 Intel Xeon 处理器为例。这个超线程技术的实现包含了两个逻辑处理器（每个都使用单独的 IA-32 架构状态来表示）。它们共享一个处理器执行引擎和总线接口。每个逻辑处理器都有它自己的高级可编程中断控制器（APIC）。

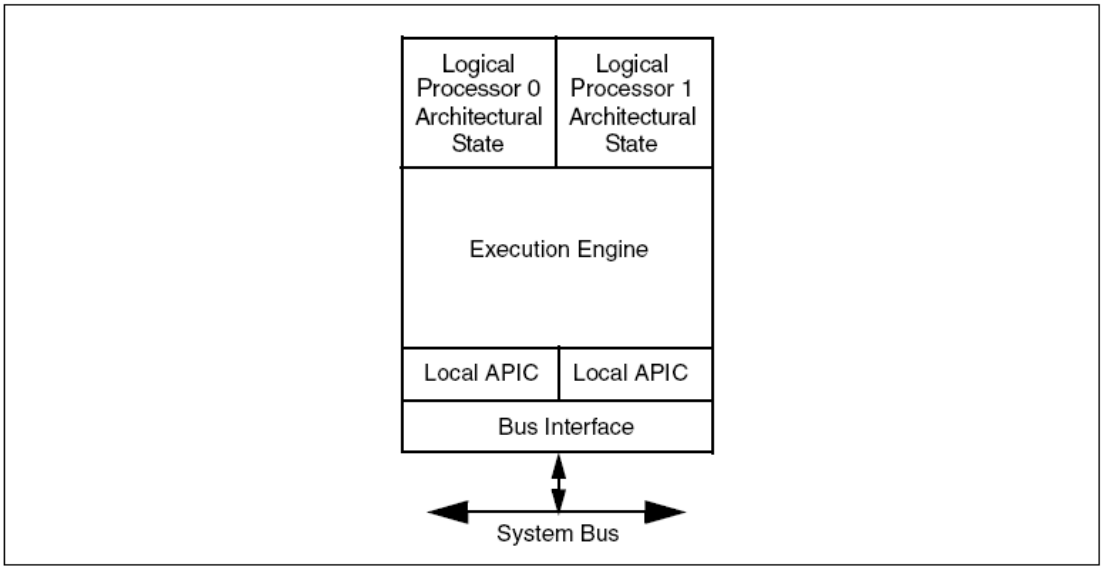

Figure 7-3. IA-32 Processor with Intel Hyper-Threading Technology using Two Logical Processors

图 7-3

使用两个逻辑处理器的带有超线程技术的 IA-32 处理器

7.6.1.1.逻辑处理器的状态

下面的特征是支持超线程技术的 IA-32 处理器的逻辑处理器架构状态的一部分。这些特征可被分成三组：

- 每个逻辑处理器都要复制的。
- 一个物理处理器中的所有逻辑处理器共享的。
- 复制或者是共享，依赖于具体实现。

下面的特征需要复制给每个逻辑处理器：

- 通用寄存器（EAX、EBX、ECX、EDX、ESI、EDI、ESP 和 EBP）。
- 段寄存器（CS、DS、SS、ES、FS 和 GS）。
- EFLAGS、EIP 寄存器。注意逻辑处理器的 CS、EIP 寄存器指向当前线程正在执行的代码流。
- x87 FPU 寄存器（ST0 到 ST7、状态字、控制字、标签字、数据操作指针、指令指针）。
- MMX 寄存器（MM0 到 MM7）。
- XMM 寄存器（XMM0 到 XMM7）和 MXCSR 寄存器。
- 控制寄存器（CR0、CR2、CR3、CR4）和系统表指针寄存器（GDTR、LDTR、IDTR、任务寄存器）。

- 调试寄存器（DR0、DR1、DR2、DR3、DR6、DR7）和调试控制 MSR。
- 机器检测全局状态（IA32_MCG_STATUS）和机器检测能力（IA32_MCG_CAP）MSR。
- 热量时钟调节和 ACPI 电源管理控制 MSR。
- 时间戳计数器 MSR。
- 大部分其它 MSR 寄存器，包括页属性表（PAT）。参看下面的例外部分。
- 本地 APIC 寄存器。

下面的特性在逻辑处理器之间共享：

- IA32_MISC_ENABLE MSR（MSR 地址 1A0H）。
- 内存类型范围寄存器（MTRR）。

下面的特性是共享还是复制依赖于具体实现：

- 机器检测架构（MCA）MSR（除了 IA32_MCG_STATUS 和 IA32_MCG_CAP MSR）。
- 性能监测控制和计数 MSR。

7.6.1.2.APIC功能性

当一个支持超线程技术的处理器初始化的时候，每个逻辑处理器都被分配一个本地 APIC ID（参看表 8-1）。本地 APIC ID 的作用就是用作逻辑处理器的 ID，存放在逻辑处理器的 APIC ID 寄存器中。如果两个或更多的支持超线程技术的 IA-32 处理器出现在一个双处理器（DP）或 MP 系统中，则系统总线上的每个逻辑处理器都会被分配一个本地 APIC ID（参看 7.7.5.“识别 MP 系统中的逻辑处理器”）。

软件利用 APIC 的处理器间中断消息机制（IPI）来与逻辑处理器通信。无论处理器是否支持超线程技术，APIC 的设立与编程都是一样的。参看第 8 章“高级可编程中断控制器（APIC）”。

7.6.1.3.内存类型范围寄存器（MTRR）

支持超线程技术的处理器的 MTRR 是在逻辑处理器中共享的。当一个逻辑处理器更新了 MTRR 的设置时，这个设置自动被同一个物理封装包中的其它逻辑处理器共享。

IA-32 架构要求所有基于 IA-32 处理器（包括逻辑处理器）的 MP 系统必须使用相同的 MTRR 内存映射。这样软件会得到一个统一的内存视图，而不管它们正在运行处理器是什么。关于如何设置 MTRR 的内容参看 10.11.“内存范围寄存器（MTRR）”。

7.6.1.4.页属性表（PAT）

每个逻辑处理器都有它自己的 PAT MSR（IA32_CR_PAT）。但是，正如 10.12.“页属性表（PAT）”中所描述的，系统中所有处理器的 PAT MSR 设置必须一致，包括逻辑处理器。

7.6.1.5.机器检测架构

在超线程技术场境内，所有的机器检测架构（MCA）的 MSR（除了 IA32_MCG_STATUS MSR 和 IA32_MCG_CAP MSR 之外）都要复制给每个逻辑处理器一份。这样在同一个物理处理器内的逻辑处理器就可以同时的初始化、配置、查询以及处理机器检测异常。这个设计和第 14 章“机器检测架构”中介绍的机器检测异常处理程序是兼容的。

IA32_MCG_STATUS MSR 是为每个逻辑处理器复制的，因此它的机器检测进行中位域标志（MCIP）可以用来检测 MCA 处理函数中的递归。另外，这个 MSR 允许每个逻辑处理器能够判断机器检测异常是否正在进行中，而不依赖于同一个物理封装中的其它逻辑处理器。

由于在同一个物理封装中的逻辑处理器被紧密地绑定在一起（因为它们共享硬件资源），它们都会收到发生在同一个物理处理器上的机器检测错误的通知。当一个严重错误发生时，如果机器检测异常被开启，所有在同一个物理处理器中的逻辑处理器都会被调度给检查异常处理程序。如果机器检测异常被关闭，则逻辑处理器进入关闭状态并发出 IERR#信号。

当开启机器检测异常时，每个逻辑处理器中 CR4 中的 MCE 标志应该被设置。

7.6.1.6.调试寄存器和扩展

每个逻辑处理器都有它自己的调试寄存器（DR0、DR1、DR2、DR3、DR6、DR7）以及它们自己的调试控制 MSR。可以分别设置这些寄存器来控制和记录每个逻辑处理器单独的调试信息。每个逻辑处理器也拥有它自己的最后分支记录栈。

7.6.1.7.性能监测计数器

性能计数器及其控制 MSR 是在一个物理处理器中的各个逻辑处理器中共享的。因此，软件可以使用这些资源。性能计数器中断、事件、以及精确事件监测以单个线程

（逻辑处理器）为基础进行建立和分配。

参看 15.11.“性能监测和超线程技术”中关于 Intel Xeon 处理器的 MP 中的性能监测的详细讨论。

7.6.1.8.IA32_MISC_ENABLE MSR

支持超线程技术的 IA-32 处理器中的 IA32_MISC_ENABLE MSR（MSR 地址 1A0H）是在各个逻辑处理器间共享的。这样，在同一个物理处理器中，这个寄存器所控制的架构特征对于各个逻辑处理器是相同的。

7.6.1.9.内存排序

支持超线程技术的 IA-32 处理器中的逻辑处理器和不支持超线程技术的处理器遵循一样的内存排序规则（参看 7.2.“内存排序”）。每个逻辑处理器使用处理器排序的内存模型，这个进一步被定义为“带有存储缓冲转发排序的写”。所有用来处理特殊编程情况而对内存排序模型进行强化和弱化的机制适用于每个逻辑处理器。

7.6.1.10.串行化指令

做为一条通用的规则，当支持 HT 的 IA-32 处理器的一个逻辑处理器执行一条串行化指令时，只有这个逻辑处理器受这条指令的影响。但是这条规则不适用于下列情况：WBINVD、INVD 和 WRMSR 指令，控制寄存器 CR0 中的 CD 标志被修改的时候执行 MOV CR 指令。这时候，两个逻辑处理器都被串行化了。

7.6.1.11.微码更新资源

在支持 HT 的 IA-32 处理器中，微码更新功能是在逻辑处理器间共享的。任何一个逻辑处理器都能够启动一次更新。每个逻辑处理器都有它自己的 BIOS 签名 MSR（IA32_BIOS_SIGN_ID，MSR 地址 8BH）。当一个逻辑处理器进行更新时，两个 IA32_BIOS_SIGN_ID MSR 被更新成相同的信息。如果两个逻辑处理器同时开始更新，处理器的内核会提供必要的同步机制来保证同一时间只能进行一个更新。

操作系统的微码更新驱动程序如果遵守了 Intel 的规范，就可以不加修改的运行在一个支持超线程技术的 IA-32 处理器上。

7.6.1.12.自修改代码

支持超线程技术的 IA-32 处理器支持自修改代码，也就是修改被缓存的或正在运行的代码。也支持交叉修改代码，即一个处理器可以修改另一个处理器缓存的或正在运行的代码。参看 7.1.3.“处理自修改和交叉修改代码”中关于 IA-32 处理器中自修改和交叉修改代码的描述。

7.6.2.实现相关的 HT 技术设施

下面的非架构性设施是与支持超线程技术的 IA-32 处理器的具体实现相关的：

- 高速缓存。
- 转换后备缓冲区（TLB）。
- 热量监测设施。

Intel Xeon 处理器的 MP 实现在后面的小节中讨论。

7.6.2.1.处理器高速缓存

对于 Intel Xeon 处理器的 MP，高速缓存是共享的。在一个逻辑处理器运行的任何高速缓存控制指令都会对所在的物理处理器的高速缓存层次体系产生全局影响。注意下面：

- **WBINVD 指令。**在将修改后的数据写回到内存后，整个高速缓存体系都置为失效。所有的逻辑处理器都停止执行，直到回写内存和失效操作执行完毕。一个特殊的总线周期被发送到所有的高速缓存代理。
- **INVD 指令。**整个高速缓存体系被置为失效，但是并不需要写回到内存中。所有的逻辑处理器都停止执行，直到失效操作执行完毕。一个特殊的总线周期被发送到所有的高速缓存代理。
- **CLFUSH 指令。**高速缓存体系中指定的高速缓存线被置为失效，但在这之前，要把修改过的数据写回内存，并把一个总线周期发送给所有的高速缓存代理，而不管是哪个逻辑处理器导致高速缓存线被填满的。
- **控制寄存器 CR0 的 CD 标志。**每个逻辑处理器都有它自己的 CR0 控制寄存器，因此也有自己的 CD 标志。两个逻辑处理器的 CD 标志以或运算的方式绑在一起的，因此其中任何一个逻辑处理器设置了 CD 标志时，则名义上整个高速缓存

就被关闭了。

7.6.2.2.处理器转换后备缓冲区（TLB）

在一个 Intel Xeon 处理器的 MP 中，数据高速缓存 TLB 是共享的，指令高速缓存 TLB 是每个逻辑处理器保存一个副本的。

用一个 ID 对 TLB 中的项做标记，表示是哪个逻辑处理器启动了这个转换。这个标记甚至用于那些标记为全局的转换，而这些转换使用了针对内存分页的页全局属性。

当一个逻辑处理器执行 TLB 失效操作时，只有被标记为这个逻辑处理器的那些项才会被刷新。这个协议适用于所有的 TLB 失效操作，包括对控制寄存器 CR3、CR4 的写操作以及 INVLPG 指令的使用。

7.6.2.3.热量监测

在一个 Intel Xeon 处理器的 MP 中，逻辑处理器间共享灾难性宕机探测器和自动热量监测机制（参看 13.16.“热量监测和保护”）。下列操作的结果也共享：

- 如果处理器的内核温度超过了预设的灾难性宕机温度，则处理器内核会停止执行，也就是两个逻辑处理器都不再运行。
- 如果处理器的内核温度超过了预设的自动热量监测跳闸温度，内核的时钟频率会自动被调整，影响两个逻辑处理器的执行速度。

对于控制时钟调整的软件来说，每个逻辑处理器都有它自己的 IA32_CLOCK_MODULATION MSR，可以开启或关闭一个处理器上的时钟调整。如果使用软件来控制时钟调整，则必须开启一个物理处理器中的所有逻辑处理器的这个功能，并且每个逻辑处理器的调整任务周期都必须置为相同的值。如果任务周期值是不同的，则处理器时钟会被调整为选择的最高任务周期。

7.6.2.4.外部信号相容性

本节讨论对从 Intel Xeon 处理器的 MP 的引脚所收到的外部信号的限制，以及这些信号是如何在它的逻辑处理器之间共享的。

- **STPCLK#。**Intel Xeon 处理器的 MP 的物理封装上提供有一个单独的 STPCLK# 引脚。外部控制逻辑利用这个引脚来控制系统的电源管理。当 STPCLK#信号被激活时，处理器内核转换到准许停止（stop-grant）状态。这个状态下，处理

器停止执行指令，但是会继续响应探听（snoop）事务。在 STPCLK#信号被激活时，无论逻辑处理器是在运行还是停止，两个逻辑处理器都停止执行，也都不再响应中断。

在 MP 系统中，通常把所有物理处理器的 STPCLK#引脚绑在一起。因此，STPCLK#信号会同时影响系统中的所有逻辑处理器。

- LINT0 和 LINT1 引脚。每个 Intel Xeon 处理器的 MP 只有一组 LINT0 和 LINT1 引脚，供逻辑处理器共享。当这些引脚中有一个被激活时，两个逻辑处理器都会响应，除非其中一个或两个逻辑处理器在它的 APIC 本地向量表中屏蔽了这个引脚。

  在 MP 系统中，典型情况下，不使用 LINT0 和 LINT1 引脚传送中断给逻辑处理器，所有中断都是通过 I/O APIC 传递给本地处理器的。

- A20M#引脚。在一个 IA-32 处理器上，典型情况下，A20M#引脚是用来与 Intel 286 处理器兼容的。激活这个引脚会导致所有对外部总线内存访问的物理地址的第 20 位被屏蔽（强制为 0）。Intel Xeon 处理器的 MP 提供一个 A20M#引脚，它对本物理处理器中的两个逻辑处理器的运行都有影响。这个配置与 IA-32 架构是兼容的。

7.6.3.探测超线程技术

软件可以使用 CPUID 指令来探测一个 IA-32 处理器是否支持超线程技术及其配置情况。以 EAX 为 1 执行 CPUID 指令，则用下面两个项报告超线程技术的情况：

- 超线程特征标志（EDX 寄存器第 28 位）表明（置位时）处理器支持超线程技术。
- EBX 的第 16 位到 23 位指明本物理封装中有多少个逻辑处理器。

当只有一个逻辑处理器可用的时候，CPUID 的超线程特征标志也可能被设置。这种情况下 EBX 的第 16 位到 23 位的值为 1。

7.6.3.1.探测是否支持MONITOR/MWAIT指令

流化（streaming）SIMD 扩展 3 引入了两条指令（MONITOR 和 MWAIT）来协助多线程软件提升线程的同步性。在最初的实现中，只有特权级为 0 的软件才能使用 MONITOR

和 MWAIT 指令。对于特权级大于 0 的（软件）可以有条件的使用这两条指令。使用下列步骤来检测 MONITOR/MWAIT 指令的可用性：

- 用 1H 调用 CPUID 来查询 MONITOR 位（ECX[3]）。如果该位为 1，那么特权级 0 就支持 MONITOR 和 MWAIT。
- 用 1H 调用 CPUID 如果报告 ECX[3]=1，则在一个 TRY/EXCEPT 异常处理中运行 MONITOR 指令。如果会导致一个运行异常。如果异常发生了，则说明在高于 0 的特权级上不支持 MONITOR 和 MWAIT。参见例 7-1。

例 7-1 验证对 MONITOR/MWAIT 的支持

```
boolean MONITOR_MWAIT_works = TRUE;
try {
   _asm {
      xor ecx, ecx
      xor edx, edx
      mov eax, MemArea
      monitor
   }
// Use monitor
} except (UNWIND) {
   // 如果能运行到这里，说明不支持MONITOR/MWAIT
MONITOR_MWAIT_works = FALSE;
```

7.6.4.初始化支持超线程技术的 IA-32 处理器

支持超线程技术的 IA-32 处理器的 MP 系统的初始化过程与传统的 MP 系统是一致的（参看 7.5.“多处理器（MP）初始化”）。系统中的一个逻辑处理器被选为 BSP，其它的处理器（或逻辑处理器）被指派为 AP。初始化过程同 7.5.3.“Intel Xeon 处理器的 MP 初始化协议算法”以及 7.5.4.“MP 初始化举例”中描述的是相同的。

作为初始化过程的一部分，每个逻辑处理器都被自动分配一个 APIC ID，存放在每个逻辑处理器的本地 APIC ID 寄存器中。如果系统中有两个或多个处理器支持超线程技术，则这些系统总线上每个逻辑处理器都被分配一个唯一的 ID（参看 7.7.5.“识别

MP 系统中的逻辑处理器”）。

一旦逻辑处理器有了 APIC ID，则软件就可以通过发送 APIC IPI 消息来与之通信。

7.6.5.在支持超线程技术的 IA-32 处理器上执行多个线程

在操作系统完成启动过程后，自举处理器（BSP）继续执行操作系统代码，其它的逻辑处理器被设置为停机状态。要想让处于停机状态的逻辑处理器执行一个代码流（线程），操作系统必须对这个逻辑处理器发送一个处理器间中断（IPI）。作为对该中断的响应，停机状态的逻辑处理器醒来（wake up）并开始执行 IPI 中断向量所指定的线程。当所有的逻辑处理器都在执行线程时，内核的执行引擎会并发的执行活动的线程。共享的执行资源按照“按需分配”（as needed basis）原则分配给活动的逻辑处理器。

为了管理在逻辑处理器上运行的多个线程，操作系统可以使用传统的对称多处理（SMP）技术。例如，操作系统可以利用时间片（time-slice）或者其它负载均衡机制来周期性的中断每个活动的逻辑处理器。一旦中断了一个逻辑处理器，操作系统检查自己的运行队列并且取出一个线程调度给这个被中断的逻辑处理器。利用这种方法，支持 MP 的操作系统能够利用同传统 MP 系统一样的调度方式在逻辑处理器上调度线程执行。

7.6.6.在支持超线程技术的 IA-32 处理器上处理中断

支持超线程技术的处理器处理中断的方式同它们在传统的 MP 系统中是一样的。APIC I/O 接收外部中断，然后作为中断消息派发给特定的逻辑处理器（图 7-4）。每个逻辑处理器也都能够通过写本地 APIC 中的 ICR 寄存器的方式给其它的逻辑处理器发送 IPI（参看 8.6.“发送处理器间中断”）。

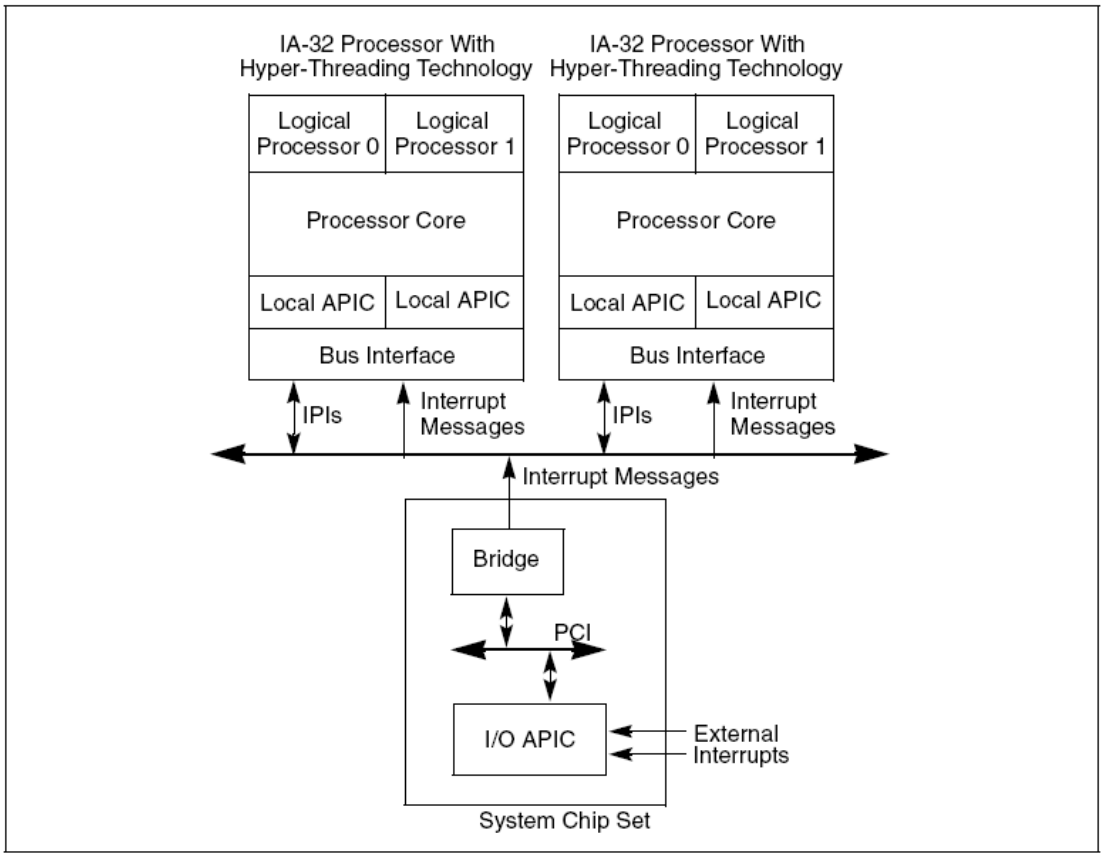

Figure 7-4. Local APICs and I/O APIC When IA-32 Processors Supporting Hyper-Threading Technology Are Used in MP Systems

图 7-4 MP 系统中使用支持超线程技术的 IA-32 处理器时的本地 APIC 和 I/O APIC

7.7.空闲和阻塞情况的管理

在支持超线程技术的 IA-32 处理器的一个逻辑处理器执行一个线程期间，逻辑处理器依照“按需分配”的原则来使用共享的处理器资源（高速缓存线、TLB 项、总线访问）。当一个处理器空闲（没有工作可做）或被阻塞（因为锁或信号量）时，可以通过使用 HLT（停止）、PAUSE、MONITOR/MWAIT 指令来完成对内核执行引擎资源的额外管理。

7.7.1.HLT 指令

HLT 指令会使调用这条指令的逻辑处理器停止执行并将其转为停机状态，直到有进一步的通知（参看《第 2 卷：指令集参考》的第 3 章“指令集参考”）。一个逻辑处理器停止后，另一个处于活动状态的逻辑处理器可以独占共享的处理器资源。被停机

的处理器使用的共享资源对活动处理器来说都是可用的，这样就大大提高了执行效率。当停机的处理器恢复执行时，共享资源又在所有的活动处理器之间共享（参看 7.7.6.2. “停止空闲的逻辑处理器”中关于如何在超线程 IA-32 处理器上使用 HLT 指令的更多描述。）

7.7.2.PAUSE 指令

在执行“自旋等待循环（spin-wait loops）”或者一个正在以密切的轮询循环（tight polling loop）访问共享锁或信号量的线程时，PAUSE 指令可以提高支持超线程技术的 IA-32 处理器的性能。在执行自旋等待循环时，处理器的性能会大大的降低，因为在退出循环的时候，处理器会检测到内存排序违例，并刷新内核处理器的流水线。PAUSE 指令可以给处理器一个暗示，表示该代码序列是一个自旋等待循环。处理器可以利用这一点来避免内存排序违例，进而避免流水线刷新。另外，PAUSE 指令会剥离流水线中的自旋等待循环，以免过多地消耗执行资源。（参看 7.7.6.1. “在自旋等待循环中使用 PAUSE 指令”中关于 HLT 指令的更多介绍。）

7.7.3.MONITOR/MWAIT 指令

操作系统通常使用空闲循环来处理线程同步。在典型的空闲循环场景中，可能有好几个忙循环，并使用了一组内存区域。一个受影响的处理器会在一个循环中等待，轮询某个内存区域来决定是否有工作要执行。典型的贴示（posting）工作就是写内存操作（等待处理器的工作队列）。启动一项工作请求并且调度好它是在那几个总线周期的序列中完成的。

从资源共享的角度来看（逻辑处理器共享执行资源），操作系统的空闲循环中使用 HLT 指令是值得的，但是会有一些问题。执行 HLT 指令使得逻辑处理器处于非执行状态。这就需要另外一个处理器（当停止的逻辑处理器的工作条件满足了）利用处理器间中断来唤醒这个停止中的处理器。这种中断的贴示和响应都会导致对新操作执行的延时。

在共享内存的配置中，特定内存区域的状态发生变化通常会导致程序从忙循环中退出；这种状态的变化通常是由另一个代理（典型情况下是另一个逻辑处理器）对该内存区域的写操作触发的。

MONITOR/MWAIT 是对 HLT 和 PAUSE 的一个补充，目的是在共享物理资源的逻辑处理器间对共享资源进行有效的分割或合并。MONITOR 建立起一个用来对写操作进行监测的有效内存地址范围；MWAIT 将处理器置于最优化状态（这可能由于实现的不同而有差异）直到有写操作作用于被监测的内存范围。

在 MONITOR 和 MWAIT 的早期实现中，只有在 CPL=0 时，它们才起作用。

两条指令都依赖处理器的监测硬件的状态。监测硬件可以是配备好的（通过执行 MONITOR 指令）或者是被触发的（由于各种不同的事件，包括对被监测的内存区域进行存储操作）。如果一旦执行了 MWAIT，则监测硬件处于被触发的状态：MWAIT 就像一条 NOP 指令一样，并继续执行指令流中的下一条指令。除非通过 MWAIT 的行为，否则无法看到监测硬件的状态。

除了对被监测内存进行写操作之外，还有其它一些事件会导致执行 MWAIT 指令的处理器被唤醒。这些包括那些会导致自愿或被迫的场境切换事件，例如：

- 外部的中断，包括 NMI、SMI、INIT、BINIT、MCERR、A20M#。
- 故障、终止（包括机器检测）。
- TLB 失效操作，包括对 CR0、CR3、CR4 和某些 MSR 的写操作；执行 LMSW（在设置了监测范围之后，但在发出 MWAIT 之前）。
- 由快速系统调用和远调用产生的自愿转换（在设置了监测范围之后，但在发出 MWAIT 之前）。

与电源管理相关的事件（例如第 2 号热量监测或者驱动 STPCLK#信号的芯片）和故障不会引起监测事件待处理标志被清除。

软件不应该允许在指令流中的 MONITOR/MWAIT 之间进行自愿的场境切换。注意，执行 MWAIT 不会重新配备好的监测硬件。也就是说 MONITOR/MWAIT 需要在一个循环中执行。还要注意的是从 MWAIT 退出可能是由于其它事件而不是对监测内存范围的写操作。软件应该自己检查一下监测区域的数据来判断是否数据已经写入。软件还应该在执行 MONITOR 指令后（在执行 MWAIT 之前）检查监测区域的地址，以确定在执行 MONITOR 指令期间有数据被写入监测区域。

提供给 MONITOR 的监测区域必须是回写高速缓存类型的。只有回写类型的内存被写入时，才能触发监测硬件。如果监测区域的内存类型不是回写的，地址监测硬件可能会被不正常的设置或者可能不会切换到配备好的状态。软件还应该保证下面的条件：

- 不想引起忙循环退出的写操作不会在监测区域中执行，

- 想引起忙循环退出的写操作一定要在监测区域中执行。

如果不保证上面的条件就会产生错误的唤醒（不是因为正确的数据写操作而导致的退出 MWAIT）。这样会导致性能的下降。软件可能会要用到 padding 来防止错误的唤醒。CPUID 既提供了判定监测区域大小的机制又提供了判定 pad 大小的机制。

7.7.4.Monitor/Mwait 地址范围判定

要想使用 MONITOR/MWAIT 指令，软件应该了解 MONITOR/MWAIT 指令监测的区域的大小，以及在多处理器系统中用于高速缓存探听的传送量的相干线尺寸（coherence line size)的大小。这个信息可以通过 CPUID 的监测叶(monitor leaf)功能(EAX=05H)查到。你将需要最大和最小监测线尺寸：

- 为了避免错过的唤醒：确保用于监测写的数据结构要适合最小的监测线尺寸。否则，在意图从 MWAIT 触发退出的写之后，处理器不可能醒过来。
- 为了避免错误的唤醒：使用最大的监测线尺寸来扩充（pad）用于监测写的数据结构。软件必须保证在数据结构之外没有无关的数据变量存在于 MWAIT 的触发区域。可能需要一个延长器（pad）来避免这种情况。

系统中，上述两个值和高速缓存尺寸是无关的，软件不应该假定它们之间有任何关系。在一个单集群系统中，这两个参数应该默认是相同的（监测触发区域的大小与系统向干线尺寸是一样的）。

基于 CPUID 返回的监测线大小，操作系统应该能够动态的分配带有适当填充（padding）的数据结构。如果操作系统要使用静态数据结构，应该修改这个结构并使用动态分配的数据缓冲区来进行线程同步。如果后者无法实现，可以考虑不在静态结构上使用 MONITOR/MWAIT 指令。

为在多集群系统上正确地设置 MONITOR/MWAIT 的数据结构：需要处理器、芯片组、BIOS 间的交互（系统相干线尺寸可能会与芯片组有关；这个尺寸可能会与处理器的监测区域的大小不同）。BIOS 负责为利用 IA32_MONITOR_FILTER_LINE_SIZE MSR 的系统相干线尺寸设置正确的值。监测区域的大小和 IA32_MONITOR_FILTER_LINE_SIZE MSR 的值经过比较后，较小的那个参数会被报告为“最小监测线尺寸”，较大的则被报告为“最大监测线尺寸”。

7.7.5.在 MP 系统中识别逻辑处理器

对于 IA-32 处理器，系统硬件在加电或复位时会建立一个初始的 APIC ID（参看 7.6.4.“初始化支持超线程技术的 IA-32 处理器”）。对于支持超线程技术的 IA-32 处理器，系统硬件会为每个连接到系统总线上的逻辑处理器分配一个唯一的 APIC ID。

逻辑处理器的 APIC ID 由三个域组成：逻辑处理器 ID，物理封装 ID 以及集群 ID。图 7-5 显示了这些域的结构。这里，第 0 位是一个 1 位的逻辑处理器 ID，第 1 位和第 2 位组成一个 2 位的封装 ID，第 3、4 位组成一个 2 位的集群 ID。第 0 位用来识别同一个封装中的两个逻辑处理器。

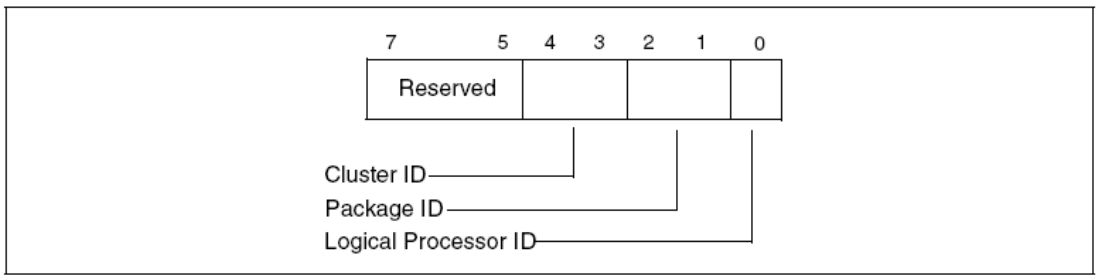

Figure 7-5. Interpretation of the APIC ID

图 7-5 APIC ID 的解释

表 7-1 显示了在一个有四类 Intel Xeon 处理器的 MP 系统（总共有 8 个逻辑处理器）中，为每个逻辑处理器产生的 APIC ID。一个 Intel Xeon 处理器的 MP 中的两个逻辑处理器中，逻辑处理器 0 被称为主逻辑处理器，逻辑处理器 1 被称为“副逻辑处理器”。

表 7-1 系统中逻辑处理器的初始 APIC ID（有四种 MP 类型的支持超线程技术的 Intel Xeon 处理器）

逻辑处理器初始 APIC ID	物理处理器 ID	逻辑处理器 ID
0H	0H	0H
1H	0H	1H
2H	1H	0H
3H	1H	1H
4H	2H	0H
5H	2H	1H
6H	3H	0H
7H	3H	1H

软件可以利用 7.5.5.“识别 MP 系统中的处理器”中介绍的两种方法中的任意一种

来确定逻辑处理器的 APIC ID。注意只有每个物理封装中的主逻辑处理器的 APIC ID 才被包含在 MP 表中。系统中的所有逻辑处理器都被包含在 ACPI 表中。主逻辑处理器在表的顶部，后面是副逻辑处理器。

在将来的支持超线程技术的 IA-32 处理器中可能会有两个的逻辑处理器，图 7-5 的逻辑处理器 ID 位将会被扩充到 2 或 3 位。封装 ID 和集群 ID 将会顺序的左移。如果封装 ID 也被扩充到多于 2 位，集群 ID 也需要左移。

用 EAX 为 1 来调用 CPUID 指令，结果放在 EBX 中，操作系统和应用程序可以通过分析 EBX 中的逻辑处理器域和 APIC 物理 ID 域来确定 APIC ID 对于特定处理器的的布局。

对于没有 HT 技术的 IA-32 处理器，软件可以通过向本地 APIC ID 写入一个值来为一个逻辑处理器分配一个 APIC ID；但是，CPUID 指令仍然会返回处理器的初始 APIC ID（在加电或复位时分配的）。

图 7-5 描述了现在的 HT 技术（有两个逻辑处理器）APIC ID 中的集群 ID、封装 ID 以及处理器 ID 的结构。一般说来，一个物理封装中的逻辑处理器中的 APIC ID（不包含集群 ID）可以表示成：

（（封装 ID << （1+（（int）（log（2）（max（每个封装中的逻辑处理器个数 -1,1））））） || 逻辑处理器 ID）

用这个公式来确定将来 HT 技术的实现中逻辑处理器以及物理处理器之间的联系。下面的伪码（例 7-1 和 7-2）显示了一个如何确定逻辑处理器和物理处理器之间关系的算法。这个算法适用于一个物理封装中有任意个逻辑处理器的情况。通过使用操作系统指定的关系（affinity）来进行绑定，这个算法在系统中的每个逻辑处理器上运行。运行了这个算法后，一个物理封装中的逻辑处理器都有一样的处理器 ID。所有的物理处理器都必须在支持同样数目的逻辑处理器。

检测 HT 技术并确定逻辑处理器与对应的物理处理器 ID 的关系的算法包含下面五个步骤：

1．检测处理器对 HT 技术的支持。

2．确定一个物理处理器封装中的逻辑处理器的个数。

3．取出这个处理器的初始 APIC ID。

4．计算一个掩码值和一个移位值。

5．计算一个逻辑处理器 ID 和一个物理处理器 ID。

Example 7-2. Generalized Algorithm to Extract Physical Processor IDs for Hyper-Threading Technology

1. Pseudo-code to detect support for Hyper-Threading Technology in a processor. 检测处理器是否支持超线程技术的伪代码。

```
// Returns non-zero if Hyper-Threading Technology is supported on
// the processors and zero if not. This does not mean that
// Hyper-Threading Technology is necessarily enabled.
// 如果支持超线程技术返回非零值，否则返回零。这并不代表超线程技术一定是开启的。
unsigned int HTSupported (void)
{
   try { // verify cpuid instruction is supported, 确定支持 cpuid 指令
      execute cpuid with eax = 0 to get vendor string
      execute cpuid with eax = 1 to get feature flag and signature
   }
   except (EXCEPTION_EXECUTE_HANDLER) {
      returen 0 ; // CPUID is not supported and so Hyper-Threading
      // Technology is not supported
   }
   // Check to see if this a a Genuine Intel Processor
   // a member of the Pentium 4 processor family
   // supporting Hyper-Threading Technology
   // 检查是否是真正 Intel 处理器，支持超线程技术
   if (vendor string NEQ GenuineIntel)
      if (family signature NEQ Pentium4Family)
         return (feature_flag_edx & HTT_BIT);
   return 0;
}
```

2. Pseudo-code to identify the number of logical processors per physical processor package.识别逻辑处理器数目的伪代码

```
#define NUM_LOGICAL_BITS 0x00FF0000 // EBX[23:16] indicate number of
                                    // logical processor per package
// Returns the number of logical processors per physical processor.
unsigned char LogicalProcessorsPerPackage (void)
{
   if (!HTSupported ( ) ) return (unsigned char) 1;
   execute cpuid with eax = 1
   store returned value of ebx
   return (unsigned char) ( (reg_ebx & NUM_LOGICAL_BITS) >> 16) ;
}
```

Example 7-3. Streamlined Determination of Mask to get the Logical Processor Number

3. Pseudo-code to extract the initial APIC ID of a processor

读取处理器初始APIC ID的伪代码

```
#define INITIAL_APIC_ID_BITS 0xFF000000 //EBX[31:24] initial APIC ID
// Returns the 8-bit unique initial APIC ID for the processor this
// code is actually running on. The default value returned is 0xFF if
// Hyper-Threading Technology is not supported.
// 返回唯一 8 位 APIC ID，默认返回值是 0xFF
unsigned char GetAPIC_ID (void)
{
   unsigned int reg_ebx = 0;
   if (!HTSupported ( ) ) return (unsigned char) -1;
   execute cpuid with eax = 1
   store returned value of ebx
   return (unsigned char) ( (reg_ebx & INITIAL_APEIC_ID_BITS) >> 24;
}
```

4. Sample code to compute a mask value and a bit-shift value, the logical processor ID and physical processor package ID.

计算掩码和移位值，以及逻辑处理器 ID、物理处理器 ID

```
unsigned char i = 1;
unsigned char PHY_ID_MASK = 0xFF;
unsigned char PHY_ID_SHIFT = 0;
unsigned char APIC_ID;
unsigned char LOG_ID、PHY_ID;
Logical_Per_Package = LogicalProcessorsPerPackage ( ) ;
While  (i < Logical_Per_Package) {
   i *= 2;
   PHY_ID_MASK <<= 1;
   PHY_ID_SHIFT++;
}
// Assume this thread is running on the logical processor from
// which we extract the logical processor ID and its physical
// processor package ID. If not、use the OS-specific affinity
// service  (See example 7-3)  to bind this thread to the target
// logical processor
// 假设这个线程正在运行在我们想从中读取 logical processor ID and its
// physical processor package ID 的逻辑处理器上。否则，请使用依赖操作系
统
// 的服务来将这个线程绑定到目标逻辑处理器上
APIC_ID = GetAPIC_ID ( ) ;
LOT_ID = APIC_ID & ~PHY_ID_MASK;
PHY_ID = APIC_ID >> PHY_ID_SHIFT;
```

Example 7-4. Using an OS-specific Affinity Service to Identify the Logical Processor Ids in an MP System

```
5. Compute the logical processor ID and physical processor
package ID.
// The OS may limit the processor that this process may run on.
// OS 可能会限制该进程运行的处理器
hCurrentProcessHandle = GetCurrentProcess ( ) ;
```

```
GetProcessAffinityMask (hCureentPorcessHandle,
&dwProcessAffinity、&dwSystemAffinity) ;
// If the available process affinity mask does not equal the
// available system affinity mask、then determining if
// Hyper-Threading Technology is enabled may not be possible.
if  (dwProcessAffinity != dwSystemAffinity)
printf  (“This process can not utilize all processors. \n”) ,
dwAffinityMask = 1;
while  (dwAffinityMask != 0 &&
dwAffinityMask <= dwProcessAffinity)  {
   // Check to make sure we can utilize this processor first.
   if  (dwAffinityMask & dwProcessAffinity) {
      if  (SetProcessAffinityMask (hCurrentProcessHandle,
            dwAffinityMask) )  {
         Sleep (0) ;  // May not be running on the logical processor
                  // on the affinity just set. Sleep gives the
                  // OS a chance to switch to the desired
               // logical processor.
         // Retrieve APIC_ID for this logical processor
         // Extract logical processor ID and physical processor
         // package ID
      }
   }
}
```

7.7.6.所需的操作系统支持

本节介绍运行在支持超线程技术的 IA-32 处理器上的操作系统要做哪些修改，以及使操作系统更有效的利用逻辑处理器的优化方法。这些修改和推荐的优化都是在 Windows XP 和 Linux 内核 2.4.0 操作系统中，针对超线程 IA-32 处理器所作的修改中

具有代表性的。其它针对支持超线程技术的 IA-32 处理器的优化在《Pentium 4 and Intel Xeon 处理器优化参考手册》中介绍（参看 1.4.“相关文献”了解其征订号）。

7.7.6.1.在自旋等待循环中使用PAUSE指令

在 Intel Xeon 或 Pentium 4 处理器中，建议在所有的自旋等待循环中使用 PAUSE 指令。

使用自旋等待循环的软件包括多处理器同步原语（自旋锁、信号量、互斥变量）和空闲循环。这种代码在等待某个资源的时候，使处理器一直执行加载——计算——分支循环。在这个循环里包含一个 PAUSE 指令会大大提高效率（参看 7.7.2.“PAUSE 指令”）。下面的代码给出了一个自旋等待循环使用 PAUSE 指令的例子：

```
Spin_Lock:
    CMP lockvar、0; Check if lock is free
    JE Get_Lock
    PAUSE ;  Short delay
    JMP Spin_Lock
Get_Lock:
    MOV EAX、1
    XCHG EAX、lockvar ;  Try to get lock
    CMP EAX、0 ;  Test if successful
    JNE Spin_Lock
Critical_Section:
    <critical section code>
    MOV lockvar、0
    ...
Continue:
```

上面的自旋等待循环使用了“测试，测试——和——设置”技术来确定同步变量的有效性。建议在书写自旋等待循环时使用这个技术。

在 Pentium 4 之前的 IA-32 处理器中，PAUSE 指令等于 NOP 指令。

在 C0 的空闲循环中使用 MONITOR/MWAIT 潜能

操作系统可能为不同的空闲状态实现了不同的处理方法。在兼容 ACPI 的操作系统

上，典型的操作系统空闲循环如例 7-5 所示：

Example 7-5. A Typical OS Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The idle loop is entered with interrupts disabled.
WHILE (1) {
   IF (WorkQueue) THEN {
      // Schedule work at WorkQueue
   } ELSE {
      // No work to do - wait in appropriate C-state handler depending
      // on Idle time accumulated
      IF (IdleTime >= IdleTimeThreshhold) THEN {
         // Call appropriate C1、C2、C3 state handler、C1 handler
         // shown below
      }
   }
}
// C1 handler uses a 停机 instruction
VOID C1Handler ()
{
STI
HLT
}
```

如果支持 MONITOR 和 MWAIT，可以考虑在 C0 空闲状态（C0 idle state loops）循环中使用它们。

Example 7-6. An OS Idle Loop with MONITOR/MWAIT in the C0 Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
```

```
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.
WHILE (1) {
   IF (WorkQueue) THEN {
      // Schedule work at WorkQueue
   } ELSE {
      // No work to do - wait in appropriate C-state handler depending
      // on Idle time accumulated
      IF (IdleTime >= IdleTimeThreshhold) THEN {
         7-44 Vol. 3
         MULTIPLE-PROCESSOR MANAGEMENT
         // Call appropriate C1、C2、C3 state handler、C1
         // handler shown below
         MONITOR WorkQueue // Setup of eax with WorkQueue LinearAddress,
         // ECX、EDX = 0
         IF (WorkQueue != 0) THEN {
            MWAIT
         }
      }
   }
}
// C1 handler uses a 停机 instruction
VOID C1Handler ()
{
STI
HLT
}
```

7.7.6.2.停止空闲逻辑处理器

如果两个逻辑处理器中的一个处于空闲状态或者在自旋等待循环中很长时间，可通过调用 HLT 指令来停止这个处理器。

在 MP 系统中，操作系统可以将空闲处理器至于循环状态中，让它们检查运行队列中是否有可以运行的任务。执行空闲循环的处理器消耗了大量的内核执行资源，这些资源本来可以共其它逻辑处理器使用。因此，将空闲的逻辑处理器停止会优化系统的性能。如果一个物理处理器中的所有逻辑处理器都被停止了，该处理器就会进入节电模式。

C1 空闲循环中使用 MONITOR/MWAIT

在 C1 空闲循环中，操作系统可以考虑用 MONITOR/MWAIT 来替换 HLT。例 7-7 列出了一个例子：

Example 7-7. An OS Idle Loop with MONITOR/MWAIT in the C1 Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
[1]// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.
WHILE (1) {
   IF (WorkQueue) THEN {
      // Schedule work at WorkQueue
   } ELSE {
      // No work to do - wait in appropriate C-state handler depending
      // on Idle time accumulated
      IF (IdleTime >= IdleTimeThreshhold) THEN {
         // Call appropriate C1、C2、C3 state handler、C1
         // handler shown below
```

[1] Excessive transitions into and out of the HALT state could also incur performance penalties. Operating systems should evaluate the performance trade-offs for their operating system.

```
        }
    }
}
// C1 handler uses a Halt instruction
VOID C1Handler ()
{
    MONITOR WorkQueue // Setup of eax with WorkQueue LinearAddress,
                      // ECX、EDX = 0
    IF （WorkQueue != 0） THEN {
        STI
        MWAIT         // EAX、ECX = 0
    }
}
```

7.7.6.3.在多个逻辑处理器上调度线程的方法

由于逻辑处理器的存在，如何在逻辑处理器上派发线程会影响系统的整体性能。建议用下面的方法来调度线程。

- 尽量给每一个物理封装中的其中一个逻辑处理器派发线程，而不是给一个物理处理器中其余剩下的逻辑处理器。在一个有多个 HT 技术的 IA-32 处理器的 MP 系统中，把线程平均分配给所有的物理处理器，而不是集中到少数几个处理器上。
- 利用处理器关联度来分配线程到一个特定的处理器，这样在线程被挂起然后再次被执行的时候，该处理器高速缓存中很可能会包含这个线程的代码。这个线程可以被派发到一个物理处理器中的两个逻辑处理器中的任何一个，因为两个逻辑处理器共享物理处理器的执行资源。

7.7.6.4.消除基于执行的计时循环

Intel 不鼓励使用利用处理器执行速度来测量时间的循环，有下面几个原因：

- 在一个处理器上校准的计时循环如果在另一个不同速度的处理器上运行就会产生问题。

- 这种基于循环的计时循环在支持超线程技术的 IA-32 处理器上运行会得到不确定的结果。原因是各个逻辑处理器共享物理处理器的执行资源。

为了避免上面的问题，计时循环就必须使用一种不依赖逻辑处理器执行速度的计时机制。下面的方法通常是可用的：

- 一个高精确度的系统计时器（例如，Intel 8254）。
- 处理器内置的高精确度计时器（例如，本地 APIC 计时器或时间戳计数器）。

更多的信息参看《Pentium 4 和Intel Xeon处理器优化参考手册》（参看1.4.“相关文献”了解征订号）。

7.7.6.5.在对齐的128字节内存块上设置锁或信号量

软件在使用锁或信号量来同步进程、线程或其它代码区域的时候，Intel 推荐在一个高速缓存线中只放置一个锁或信号量。在 Intel Xeon 处理器 MP 中（有 128 字节的高速缓存线），遵循这个建议意味着每个锁或信号量必须放在以 128 字节边界开始的 128 字节的内存块中。这样会减少锁带来的总线负荷。

第 8 章 高级可编程中断控制器（APIC）

高级可编程中断控制器（APIC）（在下述各节中指的是本地 APIC）是从 Pentium 处理器开始引入 IA-32 架构的（参看 18.24.“高级可编程中断控制器（APIC）”），在 Pentium 4、Intel Xeon 和 P6 系列处理器中都有（参看 8.4.2.“本地 APIC 的存在”）。本地 APIC 执行下述两项主要功能：

- 从处理器的中断引脚、内部资源和/或者外部 I/O APIC（或者其它外部中断控制器）接收中断，发送到处理器核心进行处理。
- 在多处理器系统（MP）中，从总线发送和接收其它 IA-32 处理器的处理器间中断（IPI）消息。这些 IPI 可用来在系统的各处理器间分发中断，或者执行广泛的系统功能（诸如，启动处理器或者在一组处理器间分发工作）。

外部 **I/O APIC** 是 Intel 的系统芯片组的一部分。它的主要功能就是从系统及其相关的 I/O 设备接收外部中断事件，并作为中断消息转发给本地 APIC。在 MP 系统中，I/O APIC 也提供一种从系统总线分发外部中断给选定的处理器或者处理器组的机制。

本章详细描述本地 APIC 和它的编程接口。也简要描述一下本地 APIC 和 I/O APIC 之间的接口。欲了解有关 I/O APIC 更详细的信息请与 Intel 联系。

当一个本地 APIC 发送一个中断到它相关的处理器核心请求处理时，处理器使用第 5 章“中断和异常处理”中描述的中断和异常处理机制来服务这个中断。5.1.“中断和异常概述”简要介绍了 IA-32 架构的中断和异常处理。为了更好地理解下面各节所介绍的 IA-32 的 APIC 架构和它的功能，建议先阅读 5.1.节。

8.1.本地 APIC 和 I/O APIC 概述

每一个本地 APIC 由一组 APIC 寄存器（参看表 8-1）和相应的硬件组成，这些硬件控制着中断往处理器核心的传送和 IPI 消息的产生。APIC 寄存器是内存映射的，可用 MOV 指令读写。

本地 APIC 可从下列来源接收中断：

- **本地连接的 I/O 设备。**这些中断是由于直接连接到处理器的本地中断引脚（LINT0 和 LINT1）的 I/O 设备激活一个边极或者电平（edge or level）引起

的。这些 I/O 设备也可以连接到 8259 之类的中断控制器上，中断控制器再通过一个本地中断引脚连接到处理器上。

- **外部连接的 I/O 设备。**这些中断是由于连接到 I/O APIC 中断引脚的 I/O 设备激活一个边极或者电平引起的。这些中断是作为 I/O 中断消息从 I/O APIC 发送到系统中的一个或多个 IA-32 处理器中的。
- **处理器间中断（IPI）。**一个 IA-32 处理器可以使用 IPI 机制通过系统总线发送中断给另外一个或者一组处理器。IPI 用于软件自我中断、中断中继或者抢先时序安排。
- **APIC 时钟产生的中断。**可对本地 APIC 时钟编程使之发送一个中断给它相关的处理器，当它到达编程设定的计数值时。
- **性能监测计数器中断。**在 Pentium 4、Intel Xeon 和 P6 系列处理器中，当有一个性能监测计数器溢出时，就可以向相关的处理器发送一个中断。（参看 15.10.6.9.“溢出时产生中断”。）
- **热量传感器中断。**在 Pentium 4 和 Intel Xeon 处理器中，当内部热量传感器跳闸时，可以发送一个中断给处理器。（参看 13.16.2.“热量监测器”。）
- **APIC 内部错误中断。**当本地 APIC 识别到一个内部错误条件时（比如试图访问一个未实现的寄存器），APIC 可被编程来发送一个中断到相应的处理器（参看 8.5.3.“错误处理”）。

这些中断源中，处理器的 LINT0 和 LINT1 引脚、APIC 时钟、性能监测计数器、热量传感器和内部 APIC 错误探测器被称作**本地中断源**。收到本地中断源信号时，本地 APIC 使用本地中断传送协议来传送中断到处理器核心。本地中断传送协议是通过一组称作**本地向量表**或者 **LVT**（参看 8.5.1.“本地向量表”）的 APIC 寄存器来设置的。每个本地中断源在本地向量表中都有一个单独的项，这样，每个中断源都可以设立一个专门的中断传送协议。比如，如果打算把 LINT1 引脚用作 NMI 引脚，那么，可以设置本地向量表中的 LINT1 项，使之传送 2 号向量（NMI 中断）到处理器核心。

本地 APIC 是通过 IPI 消息处理设施来处理其余两种中断源（外部连接的 I/O 设备和 IPI）的。

可以通过对处理器的本地 APIC 的中断命令寄存器（ICR）编程，来让它产生 IPI（参看 8.6.1.“中断命令寄存器（ICR）”）。对 ICR 的写操作导致一个 IPI 消息的产生，和在系统总线（对 Pentium 4 和 Intel Xeon 处理器而言）或者 APIC 总线（对 Pentium

和 P6 系列处理器而言）上的发送。

（参看 8.2.“系统总线和 APIC 总线的对比”。）IPI 可被发给系统中的其它处理器或者产生它的处理器（自我中断）。当目标处理器收到一个 IPI 消息时，它的本地 APIC 自动处理这个消息（使用包含在这个消息中的信息比如向量号和触发模式）并把它传送到处理器核心请求服务。有关本地 APIC 的 IPI 消息的传送和接受机制的详细解释，参看 8.6.“发出处理器间中断”。

本地 APIC 也可通过 I/O APIC 接收外部连接的设备的中断（参看图 8-1）。I/O APIC 负责接收系统硬件和 I/O 设备产生的中断，并负责把它们作为中断消息转发给本地 APIC。

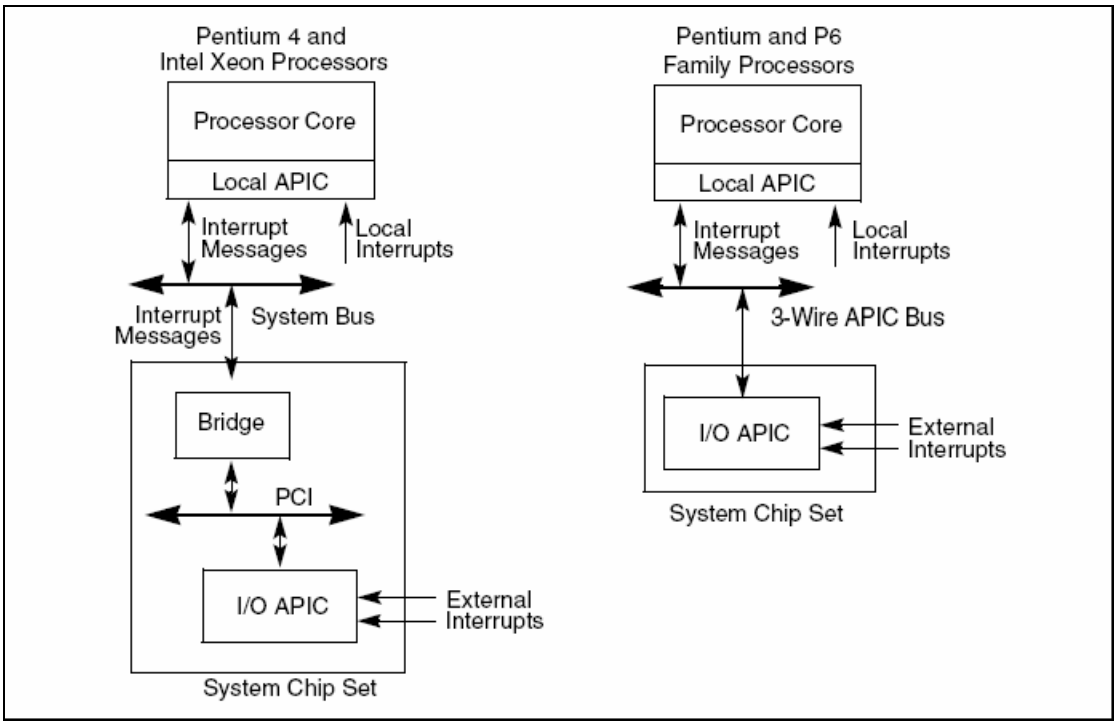

图 8-1 单处理器系统中本地 APIC 和 I/O APIC 的关系

I/O APIC 上的每一个单个引脚都可被编程，来在被激活时产生一个特定的中断向量。I/O APIC 也有一个“虚拟电线模式”，可与标准的 8259A 类型的外部中断控制器进行通信。

注意，本地 APIC 可被关闭（参看 8.4.3.“开启或关闭本地 APIC”），以便它相关的处理器核心直接从 8259A 中断控制器接收中断。

本地 APIC 和 I/O APIC 都可在 MP 系统中运行（参看图 8-2）。此时，每个本地 APIC 既要处理来自 I/O APIC 的作为中断消息的外部产生的中断和来自系统总线上其它处理器的 IPI，还要处理来自自身的中断。（中断也可以通过本地中断引脚传送给各自的处

理器，但是，这不是 MP 系统中普遍使用的机制。）

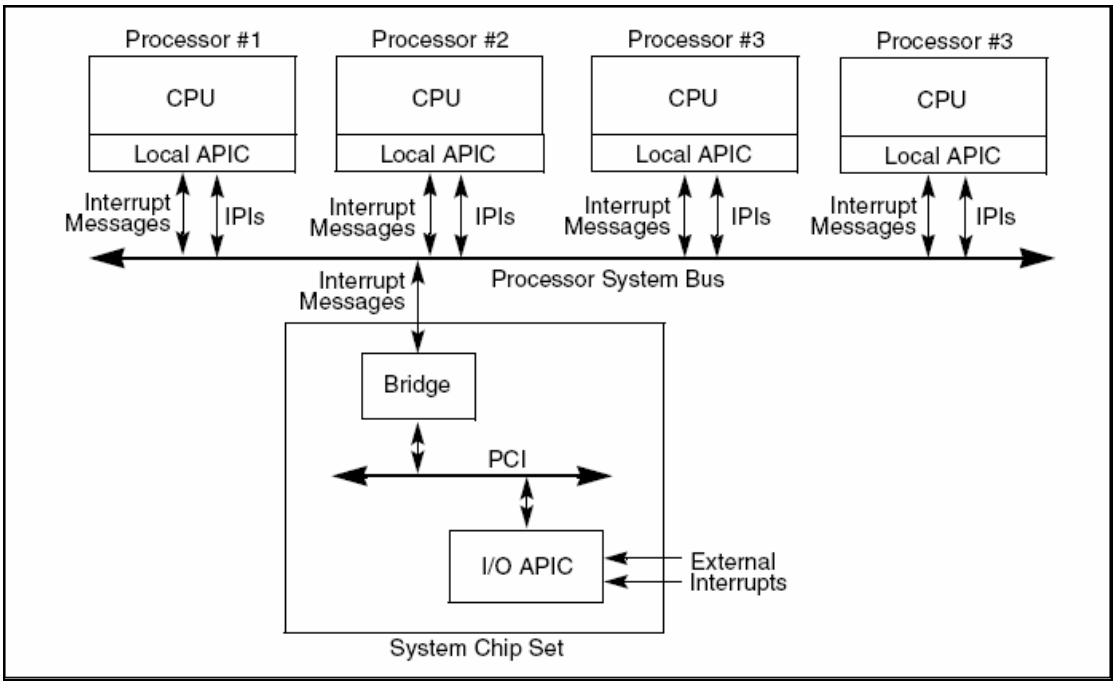

图 8-2 使用 Intel Xeon 处理器的多处理器系统中的本地 APIC 和 I/O APIC

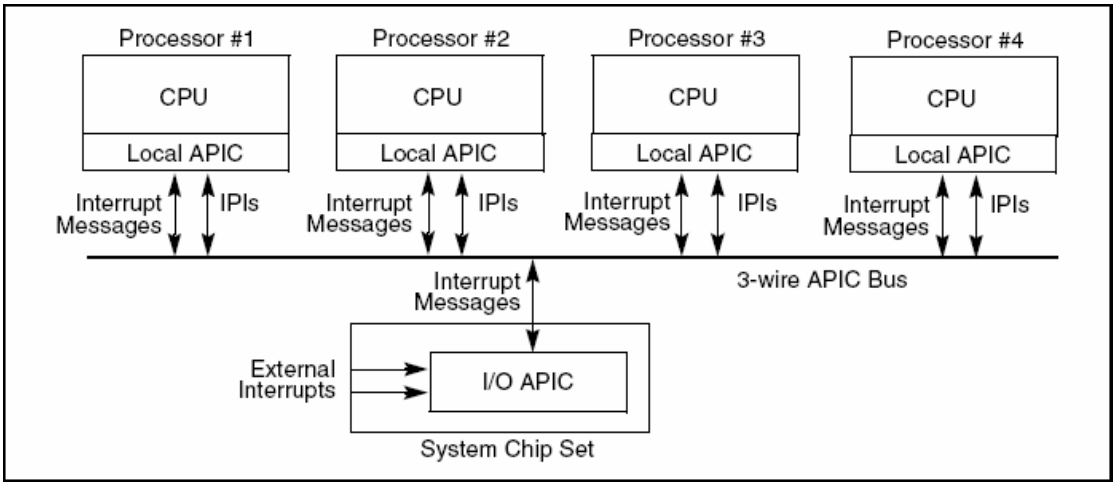

图 8-3 使用 P6 系列处理器的多处理器系统中的本地 APIC 和 I/O APIC

典型情况下，IPI 机制用在 MP 系统中通过系统总线发送固定中断（有一个专门中断号的中断）和特殊目的中断给其它处理器，或者给它自己。例如，一个本地 APIC 可以使用一个 IPI 转发一个固定中断给另外一个处理器请求服务。特殊目的的 IPI，包括 NMI、INIT、SMI 和 SIPI IPI，允许总线上的一个或者多个处理器执行系统范围内的引导和控制功能。

下面各节专门讲解本地 APIC 及其在 Pentium 4、Intel Xeon 和 P6 系列处理器中的实现。术语“本地 APIC”和“I/O APIC”指的是用在 P6 系列处理器中的局部 APIC

和 I/O APIC，以及用在 Pentium 4、Intel Xeon 处理器中的局部 APIC 和 I/O xAPIC（参看 8.3.“Intel 82489DX 外部 APIC、APIC 和 xAPIC 之间的关系”）。

8.2.系统总线与 APIC 总线的对比

对于 P6 系列和 Pentium 处理器来说，I/O APIC 和本地 APIC 是通过 3 线（3-wire）的 APIC 间的总线通信的。本地 APIC 也使用 APIC 总线来发送和接收 IPI。对于软件来说，APIC 总线及其消息是不可见的，并不归属于架构。

从 Pentium 4 和 Intel Xeon 处理器开始，I/O APIC 和本地 APIC（使用 xAPIC 架构）通过系统总线通信（参看图 8-2）。这里，I/O APIC 通过桥硬件在系统总线上发送中断请求给处理器，桥是 Intel 芯片组的组成部分。是桥硬件产生实际的中断消息送往本地 APIC。本地 APIC 之间的 IPI 直接在总线上传输。

8.3.Intel 82489DX 外部 APIC、APIC 和 xAPIC 之间的关系

P6 系列和 Pentium 处理器的本地 APIC 是 Intel 82489DX 外部 APIC 的架构子设备。它们之间的差异在 18.24.1.“本地 APIC 和 82489DX 之间的软件可见差异”中描述。

Pentium 4 和 Intel Xeon 处理器的 APIC 架构（称作 xAPIC 架构）是 P6 系列处理器中建立的 APIC 架构的扩展。主要区别在于:使用 xAPIC 架构时,本地 APIC 和 I/O APIC 之间是通过系统总线通信的；而使用 APIC 架构时，它们是通过 APIC 总线通信的（参看 8.2.“系统总线和 APIC 总线的对比”）。而且，在 xAPIC 架构中，APIC 架构的某些特征被扩展或者修改了。这些将在下面各节中说明。

8.4.本地 APIC

本节描述本地 APIC 架构及如何探测、识别它，确定它的状态。8.5.1.“本地向量表”和 8.6.1.“中断命令寄存器（ICR)”描述如何对本地 APIC 编程。

8.4.1.本地 APIC 块图

图 8-4 给出了本地 APIC 的功能块图。软件通过读和写本地 APIC 的寄存器来与之交互作用。APIC 寄存器是映射到内存的处理器物理地址空间中的 4K 字节区域，初始启

动地址为 FEE00000H。对于正确的 APIC 操作而言，这个地址空间必须被映射到一块内存区域，且该区域绝对不可高速缓存（UC）。

在 MP 系统配置中，系统总线上的所有 IA-32 处理器的 APIC 寄存器最初都被映射到同一块 4K 字节的物理地址空间中。软件有两个选择，或者将所有的本地 APIC 的这个最初映射改变到一个不同的 4K 字节区域中，或者将每个本地 APIC 的 APIC 寄存器映射到它自己的 4K 字节区域中。如何为特定的处理器重新分配 APIC 寄存器的基地址，参看 8.4.5.“重新分配本地 APIC 寄存器”中的描述。

注意

对 Pentium 4、Intel Xeon 和 P6 系列处理器来说，APIC 在内部处理所有对 4K 字节 APIC 寄存器空间地址的内存访问，不产生外部总线周期。对于具有内建 APIC 芯片的 Pentium 处理器而言，要产生总线周期来访问 APIC 寄存器空间。因而，对于要运行在 Pentium 处理器上的软件来说，系统软件不应该显式地映射 APIC 寄存器空间到正规的系统内存中。如此做可能会导致产生非法操作码异常（#UD）或者不可预测的执行后果。

表 8-1 显示了 APIC 寄存器是如何映射到 4K 字节的 APIC 寄存器空间的。所有寄存器的长度都是 32 位、64 位或者 256 位的，并且都是 128 位边界对齐的。所有 32 位寄存器都要用 128 位边界对齐的 32 位加载或者保存来进行访问。较长的寄存器（64 位或者 256 位）必须用多个 32 位加载或者保存来进行访问，同时第一次访问要 128 位对齐。用 MOV 指令访问 APIC 地址空间时如果有 LOCK 前缀，则忽略这个前缀。也就是说，加锁操作不会生效。表 8-1 列出的所有寄存器将在本章后续各节进行描述。

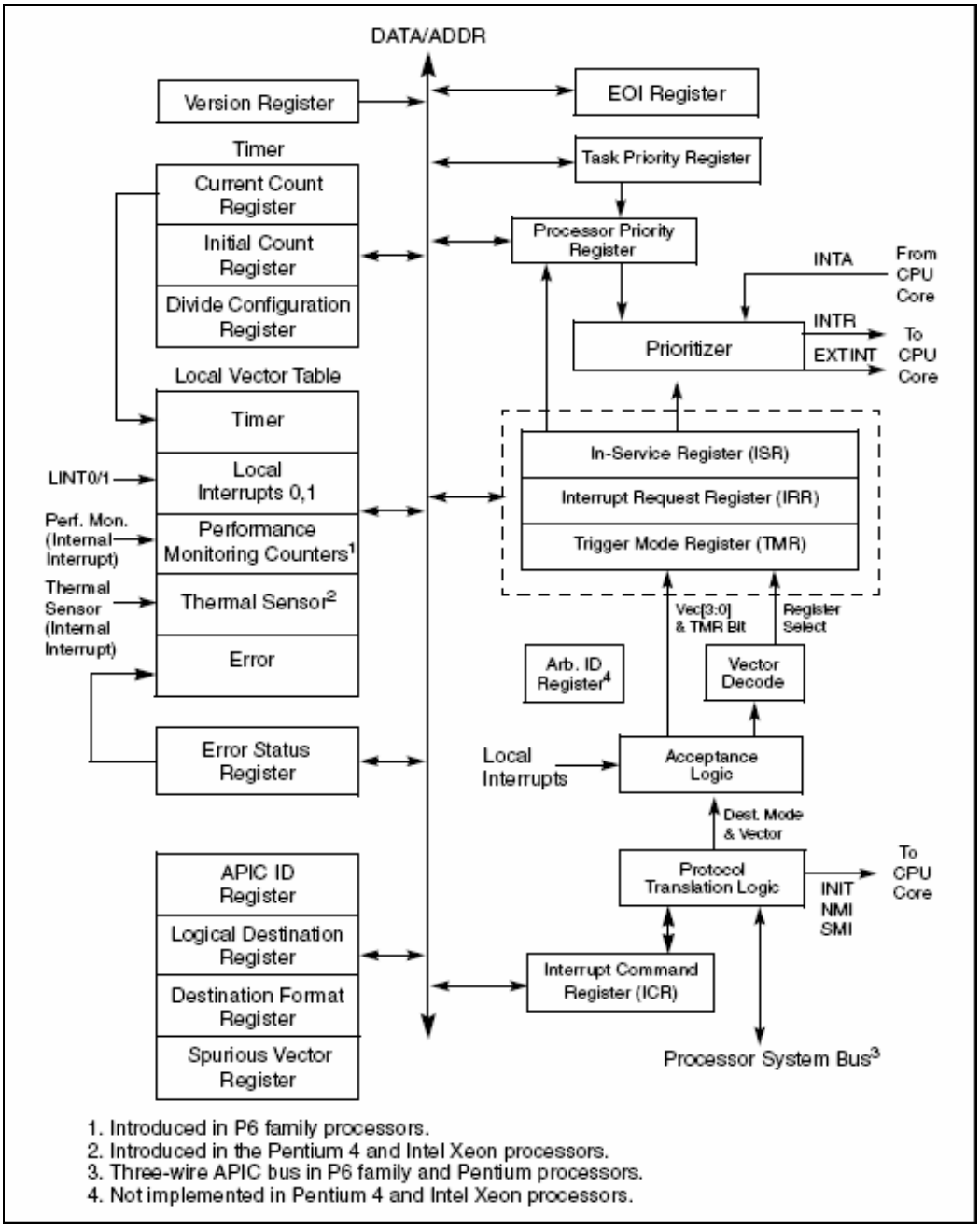

图 8-4 本地 APIC 结构

表 8-1 本地 APIC 寄存器地址映像

地　址	寄存器名称	软件读/写
FEE0 0000H	保留	
FEE0 0010H	保留	
FEE0 0020H	本地 APIC ID 寄存器	读/写
FEE0 0030H	本地 APIC 版本寄存器	只读
FEE0 0040H	保留	
FEE0 0050H	保留	
FEE0 0060H	保留	

FEE0 0070H	保留	
FEE0 0080H	任务优先权寄存器（TPR）	读/写
FEE0 0090H	仲裁优先权寄存器（APR）	只读
FEE0 00A0H	处理器优先权寄存器（PPR）	只读
FEE0 00B0H	EOI 寄存器	只写
FEE0 00C0H	保留	
FEE0 00D0H	逻辑目的寄存器	读/写
FEE0 00E0H	目的格式寄存器	第 0-27 位只读，第 28-31 位读/写
FEE0 00F0H	伪中断向量寄存器	第 0-8 位读/写，第 9-31 位只读
FEE0 0100H 到 FEE0 0170H	在服务寄存器（ISR）	只读
FEE0 0180H 到 FEE0 01F0H	触发模式寄存器（TMR）	只读
FEE0 0200H 到 FEE0 0270H	中断请求寄存器（IRR）	只读
FEE0 0280H	错误状态寄存器	只读
FEE0 0290H 到 FEE0 02F0H	保留	
FEE0 0300H	中断命令寄存器（ICR）[0-31]	读/写
FEE0 0310H	中断命令寄存器（ICR）[32-63]	读/写
FEE0 0320H	LVT 时钟寄存器	读/写
FEE0 0330H	LVT 热量传感器寄存器[2]	读/写
FEE0 0340H	LVT 性能监测计数器寄存器[3]	读/写
FEE0 0350H	LVT LINT0 寄存器	读/写
FEE0 0360H	LVT LINT1 寄存器	读/写
FEE0 0370H	LVT 错误寄存器	读/写
FEE0 0380H	初始计数寄存器（针对计时器）	读/写
FEE0 0390H	当前计数寄存器（针对计时器）	只读
FEE0 03A0H 到 FEE0 03D0H	保留	
FEE0 03E0H	除配置寄存器（针对计时器）	读/写
FEE0 03F0H	保留	

备注：

1. Pentium 4 和 Intel Xeon 处理器中不支持。

2. Pentium 4 和 Intel Xeon 处理器中引入。这个 APIC 寄存器及其相关功能是与处理器具体实现相关的，并且可能不会出现在未来的 IA-32 处理器中。

3. Pentium Pro 处理器中引入的。这个 APIC 寄存器及其相关功能是与处理器具体实现相关的，并且可能不会出现在未来的 IA-32 处理器中。

注意

表 8-1 中所列出的本地 APIC 寄存器不是 MSR。与本地 APIC 编程相关的唯一 MSR 是

IA32_APIC_BASE MSR（参看 8.4.3.“开启和关闭本地 APIC”）。

8.4.2.本地 APIC 的存在

从 P6 系列处理器开始，可以用 CPUID 指令探测内建的本地 APIC 是否存在。当用 EAX 寄存器为 1 的作为源操作数执行 CPUID 指令时，返回在 EDX 寄存器中的 CPUID 特征标志的第 9 位指出本地 APIC 是存在（置位）还是不存在（清除）。

8.4.3.开启或关闭本地 APIC

用下述两种方法之一可以开启或者关闭本地 APIC：

- 使用 IA32_APIC_BASE MSR 中的 APIC 全局开启/关闭标志（参看图 8-5）。
- 使用伪中断向量寄存器中的 APIC 软件开启/关闭标志（参看图 8-22）。

IA32_APIC_BASE MSR 中的 APIC 全局开启/关闭标志可以永久性关闭本地 APIC。加电或复位之后，这个标志是置位的，开启本地 APIC。软件可以清除这个标志来永久性关闭本地 APIC，直到下次加电或复位（来恢复它）。

当这个标志清除时，处理器在功能上就等于没有内建的 APIC（比如 Intel 486 处理器）。此时，CPUID 特征标志的 APIC 相应位（EDX 寄存器的第 9 位[参看 8.4.2.“本地 APIC 的存在”]）置为 0。而且，当 IA32_APIC_BASE MSR 中的 APIC 全局开启/关闭标志被清除时，只有加电或复位才可恢复它。

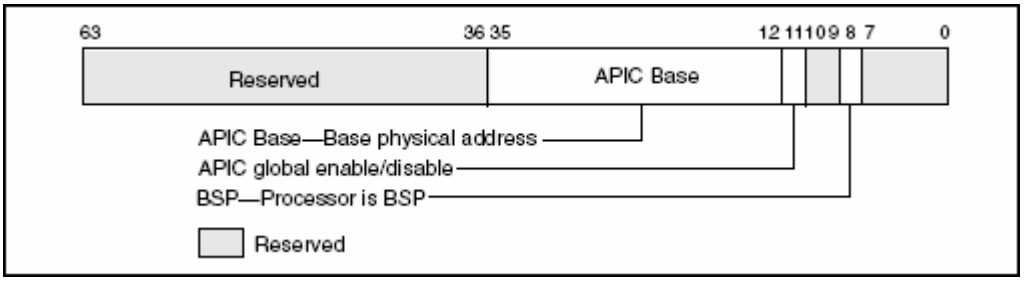

图 8-5 IA32_APIC_BASE MSR

对 Pentium 处理器而言，加电和复位期间使用 APICEN 引脚（是和 PICD1 引脚共享的）来关闭本地 APIC。

如果 IA32_APIC_BASE MSR 中的 APIC 全局开启/关闭标志未被清除，则软件可以随时通过清除伪中断向量寄存器中的 APIC 软件开启/关闭标志（参看图 8-22）来暂时关闭本地 APIC。软件关闭时本地 APIC 的状态参看 8.4.7.2.“软件关闭后本地 APIC 的状

态”中的描述。当本地 APIC 处于软件关闭状态中时，可以随时通过设置 APIC 软件开启/关闭标志来重新开启。

注意，LVT 中的每个项都有一个屏蔽位，可用来禁止中断从选定的本地中断源（LINT0 和 LINT1 引脚、APIC 时钟、性能监测计数器、热量传感器或者内部 APIC 错误探测器）传送到处理器。

8.4.4.本地 APIC 状态和位置

本地 APIC 的状态和位置保存在 IA32_APIC_BASE MSR（在 P6 系列处理器中叫做 APIC_BASE_MSR）中。这个 MSR 位于 MSR 的第 27 个地址处（1BH）。图 8-5 显示了这个 MSR 位的编码。这些位的功能如下：

BSP 标志，第 8 位 指出处理器是否自举处理器（BSP）（参看 7.5.“多处理器（MP）初始化”）。随着加电或复位，被选为 BSP 的处理器会置位这个标志，而余下的每个应用处理器（AP）把它置为 0。

APIC 全局开启标志，第 11 位 开启（1）或者关闭（0）本地 APIC（参看 8.4.3.“开启或关闭本地 APIC”）。这个标志在 Pentium 4、Intel Xeon 和 P6 系列处理器中可用。不保证在未来的 IA-32 处理器中可用或者相同的位置可用。

APIC 基域，第 12 到 35 位 指定 APIC 寄存器的基地址。把这 24 位值向低端扩展 12 位就形成了基地址，自动是 4K 字节边界对齐的。在加电或复位以后，这个域的值是 FEE00000H。

IA32_APIC_BASE MSR 中的第 0 到 7 位、第 9 和 10 位、第 36 到 63 位是保留的。

8.4.5.重新分配本地 APIC 寄存器

在 Pentium 4、Intel Xeon 和 P6 系列处理器中，可以通过修改 IA32_APIC_BASE MSR 中的 24 位基地址域中的值，来改变 APIC 寄存器的起始地址从 FEE00000H 到另外一个物理地址。这种 APIC 架构上的扩展用来解决与现存系统的内存映像的冲突，并允许 MP 系统中各个处理器映射它们自己的 APIC 寄存器到物理内存的一个不同的地方。

8.4.6.本地 APIC ID

加电的时候,系统硬件指派一个唯一的 APIC ID 给系统总线(对 Pentium 4 和 Intel Xeon 处理器而言)或者 APIC 总线(对 Pentium 和 P6 系列处理器而言)上每个本地 APIC。这个硬件指派的 APIC ID 是基于系统拓扑结构的，并且包含插槽位置和集群信息的解码信息。

在 MP 系统中，通过 BIOS 和操作系统，本地 APIC ID 也用作处理器 ID。但是，软件能否修改 APIC ID 是与模型相关的。因此，操作系统软件应该避免写本地 APIC ID 寄存器。

通过从 A11#、A12#和 BR0#到 BR3#等引脚（对 Pentium 4、Intel Xeon 和 P6 处理器而言）和 BE0#到 BE3#引脚（对 Pentium 处理器而言）中取样，处理器接收硬件指派的 APIC ID。从这些引脚取得的 APIC ID 保存在本地 APIC ID 寄存器的 APIC ID 域中（参看图 8-6),并作为处理器的初始 APIC ID。也是用 EAX 寄存器为 1 的源操作数执行 CPUID 指令时返回到 EDX 寄存器中的值。

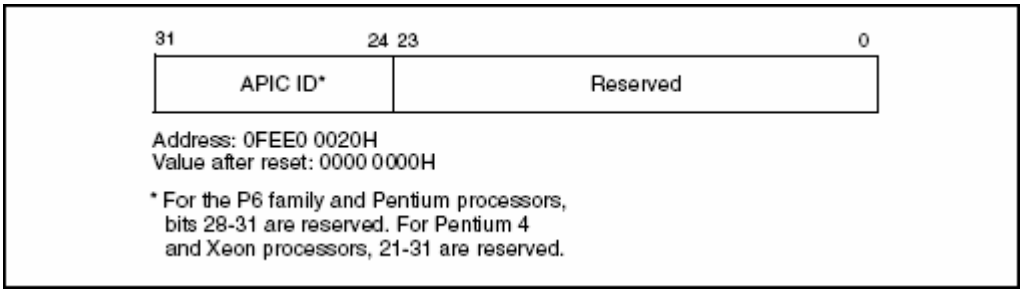

图 8-6 本地 APIC ID 寄存器

对 P6 系列和 Pentium 处理器而言，本地 APIC ID 寄存器的 APIC ID 域是 4 位，编码 0H 到 EH 可用来唯一地标识连接到 APIC 总线上的 15 个不同的处理器。对 Pentium 4 和 Intel Xeon 处理器来说，xAPIC 规格扩展本地 APIC ID 域到 8 位，可用来在系统标识 255 个处理器。

8.4.7.本地 APIC 状态

下面各小节描述本地 APIC 及其寄存器的状态，包括加电或复位之后，被软件关闭之后，一个 INIT 复位之后，和一个未激活的 INIT 消息的复位之后。

8.4.7.1.加电或复位之后本地APIC的状态

处理器加电或复位之后，本地 APIC 及其寄存器的状态如下：

- 下列寄存器复位为全 0：IRR、ISR、TMR、ICR、LDR 和 TPR 寄存器；计时器初始计数和计时器当前计数寄存器；划分配置寄存器。
- DFR 寄存器都复位成 1。
- LVT 寄存器中所有项中的屏蔽位都复位成 1，其它位都复位成 0。
- 本地 APIC ID 寄存器设成唯一的 APIC ID。（仅对 Pentium 和 P6 系列处理器而言）Arb ID 寄存器设成 APIC ID 寄存器中的值。
- 伪中断向量寄存器初始化成 0000 00FFH。第 8 位设成 0，即软件关闭本地 APIC。
- 如果系统中只有这个唯一的处理器或者是 MP 中被指定为 BSP 的处理器（参看 7.5.1.“BSP 和 AP 处理器”），本地 APIC 将对 INIT 和 NMI 消息和 INIT#和 STPCLK#信号作出正常反应；如果是 MP 中被指定为 AP 的处理器，本地 APIC 地反应同上面一样，此外，还将对 SIPI 消息作出反应。对 P6 系列处理器来说，AP 将不对 STPCLK#信号做出反应。

8.4.7.2.被软件关闭之后本地APIC的状态

当伪中断向量寄存器中的 APIC 软件开启/关闭标志被显式地清除时（与加电或复位期间的清除相对），本地 APIC 暂时被关闭（参看 8.4.3.“开启或者关闭本地 APIC”）。在这个软件关闭期间，本地 APIC 的操作和反应如下：

- 本地 APIC 将对 INIT、NMI、SMI 和 SIPI 消息作出正常反应。
- IRR 和 ISR 寄存器中的待处理中断被 CPU 保持着，请求屏蔽或者处理。
- 本地 APIC 依然可以发出 IPI。如果不希望通过 IPI 机制发送中断，那么，应该由软件负责避免 IPI 通过这种机制和 ICR 寄存器发出。
- 当本地 APIC 被关闭时，如果有在途 IPI，则在它们的接收和传送完成之后，本地 APIC 才进入软件关闭状态。
- 所有针对 LVT 项的屏蔽位都置位。忽视所有复位这些位的企图。
- （Pentium 和 P6 系列处理器）本地 APIC 继续监听所有总线消息，保持它的仲裁 ID 与系统的其余部分同步。

8.4.7.3.一个INIT复位之后本地APIC的状态

可通过下列两种方式启动处理器的 INIT 复位：

- 激活处理器的 INIT#引脚。
- 发给处理器一个 INIT IPI（即发送一个传送模式设置成 INIT 的 IPI）。

一旦通过上述两种途径收到一个 INIT，处理器即以开始本处理器核心和本地 APIC 的初始化过程作为响应。INIT 复位之后的本地 APIC 的状态，除了对 APIC ID 和仲裁 ID 寄存器没有影响之外，与加电或硬件复位之后的状态相同。这个状态也被称作“等待 SIPI”状态。参看 7.5.2.“Intel Xeon 处理器的 MP 初始化协议需求和限制”。

8.4.7.4.收到一个未激活的INIT的IPI之后本地APIC的状态

（只针对支持未激活的 INIT 的 IPI 的 Pentium 和 P6 处理器。）一个未激活的 INIT 的 IPI，除了在仲裁 ID 寄存器中加载 APIC ID 寄存器的值之外，对 APIC 的状态没有影响。

8.4.8.本地 APIC 版本寄存器

本地 APIC 包含一个硬连线的版本寄存器，软件可使用它来识别 APIC 版本（参看图 8-7）。此外，这个寄存器指定特定寄存器实现中的 LVT 中的项数。本地 APIC 版本寄存器中的域如下：

版本　　本地 APIC 的版本号：

　　1XH　　本地 APIC。Pentium 4 和 Intel Xeon 处理器中返回 14H。

　　0XH　　82489DX 外部 APIC。

　　20H 到 FFH　　保留。

最大 LVT 项　　LVT 项数减 1。对于 Pentium 4 和 Intel Xeon 处理器来说这个域的值是 5（它们有 6 个 LVT 项）；P6 系列处理器，这个值是 4（它有 5 个 LVT 项）；Pentium 处理器，这个值是 3（它有 4 个 LVT 项）。

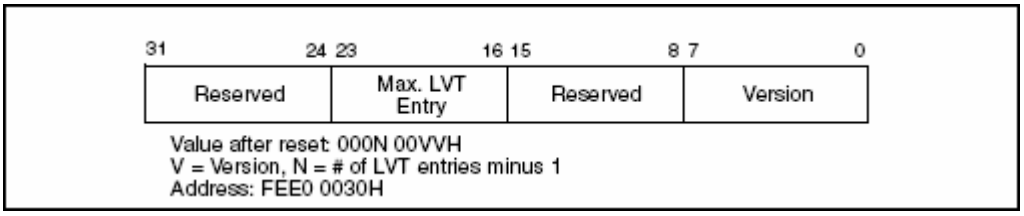

图 8-7 本地 APIC 版本寄存器

8.5.处理本地中断

下面各小节描述本地 APIC 提供的处理中断的设施，包括处理器的 LINT0 和 LINT1 引脚、APIC 计时器、性能监测计数器、热量传感器、内部 APIC 错误探测器。本地中断处理设施包括 LVT、错误状态寄存器（ESR）、划分配置寄存器（DCR）、初始计数和当前计数寄存器。

8.5.1.本地向量表

本地向量表（LVT）允许软件指定本地中断传送到处理器核心的方式。它由下列 5 个 32 位的 APIC 寄存器组成，每个本地中断使用一个：

- LVT 计时器寄存器（FEE0 0320H）——当 APIC 计时器发出一个中断信号时，指定中断传送（参看 8.5.4.“APIC 计时器”）。
- LVT 热量监测寄存器（FEE0 0330H）——当热量监测传感器产生一个中断时，指定中断传送（参看 13.16.2.“热量监测”）。这个 LVT 项是与实现相关的，不是架构上的。如果安装了它，基地址总是 FEE0 0330H。
- LVT 性能计数器寄存器（FEE0 0340H）——在溢出过程中当性能计数器产生一个中断时，指定中断传送（参看 15.10.6.9.“溢出时产生中断”）。这个 LVT 项是与实现相关的，不是架构上的。如果安装了它，基地址总是 FEE0 0340H。
- LVT LINT0 寄存器（FEE0 0350H）——当 LINT0 引脚发出中断时，指定中断传送。
- LVT LINT1 寄存器（FEE0 0360H）——当 LINT1 引脚发出中断时，指定中断传送。
- LVT 错误寄存器（FEE0 0370H）——当 APIC 探测到一个内部错误时，指定中断传送（参看 8.5.3.“错误处理”）。

注意

LVT 性能计数器寄存器和它相关的中断是在 P6 处理器中引入的，在 Pentium 4 和 Intel Xeon 处理器中也存在。LVT 热量监测寄存器和它相关的中断是在 Pentium 4 和 Intel Xeon 处理器中引入的。

注意图 8-8 所示，对某些项来说，这些域和标志中的某些是不可用的（保留）。

LVT 表中寄存器的设置信息可按如下方式指定：

向量 中断向量号

传送模式 指定发送给处理器的中断类型。注意，当与某个特定的触发模式共同使用时，某些传送模式只是操作意向。允许的传送模式如下：

000（固定的） 传送向量域中指定的中断。

010（SMI） 通过处理器的本地 SMI 信号路径传送一个 SMI 中断给处理器核心。当使用这个传送模式时，为了与未来兼容，向量域应该设成 00H。

100（NMI） 传送一个 NMI 中断给处理器。忽略向量信息。

101（INIT） 传送一个 INIT 请求给处理器核心，使之执行一次初始化。当使用这个传送模式时，为了与未来兼容，向量域应该设成 00H。

111（ExtINT） 使处理器对这个中断做出响应，就好像这个中断源自一个外部连接着的中断控制器（8259A 兼容的）。与 ExtINT 相应的一个特殊总线周期被送给这个外部中断控制器。预计由外部中断控制器提供向量信息。一个系统的 APIC 架构只提供一个 ExtINT 源，通常包含在兼容性桥中。

传送状态（只读） 指明中断传送状态，具体如下：

0（空闲） 这个中断源当前没有活性，或者该中断源以前的中断被送到了处理器核心并被接受。

1（待发送） 指明来自这个中断源的中断已经被传送到处理器核心，但是，还没被接受（参看 8.5.5.“本地中断接受”）。

中断输入 指定相应中断引脚的极性：（0）活性高，或者（1）活性低。

引脚极性

远程 IRR 标志（只读） 对于固定模式的电平触发（level-triggered）的中断来说，当本地 APIC 接到中断服务请求时，此标志置位；当从处理器收到 EOI 命令时，此标志复位。对于边极触发（edge-triggered）和其它模式来说，此标志未定义。

触发模式 为本地 LINT0 和 LINT1 引脚选择触发模式：（0）边极感应的（1）电平感应的。仅当传送模式固定时，才使用这个标志。当传送模式是 NMI、SMI 或者 INIT 时，触发模式通常是边极感应的；当传送模式是 ExtINT 时，触发模式总是电平感应的。计时器和错误中断总被当作是边极感应的。

如果本地 APIC 不是随同 I/O APIC 一起使用的，并且选择了固定模式，则 Pentium 4、Intel Xeon 和 P6 系列处理器将总是使用电平感应触发，而不管是否选择了边极感应触发。

屏蔽 中断屏蔽：（0）开启中断接受（1）禁止中断接受。当本地 APIC 处理一个性能监测计数器中断时，它会自动设置相应 LVT 项中的屏蔽位。这个标志将保持到软件清除为止。

计时器模式 选择计时器模式：（0）一次性的（1）周期性的（参看 8.5.4.“APIC 计时器”）。

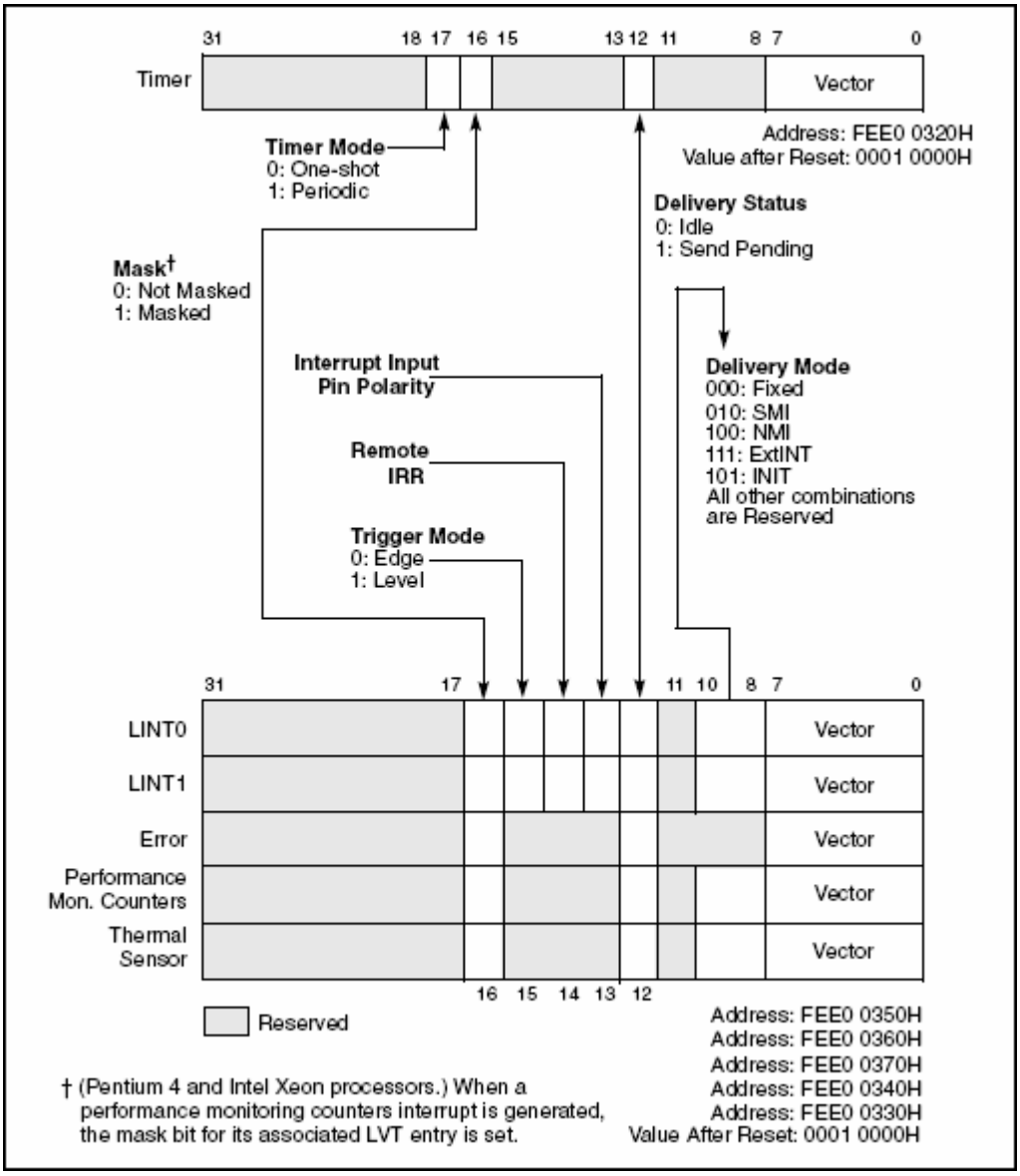

图 8-8 本地向量表（LVT）

8.5.2. 合法中断向量

IA-32 架构定义了 256 个中断向量，从 0 到 255（参看 5.2.“异常和中断向量”）。本地 APIC 和 I/O APIC 支持其中的 240 个（从 16 到 255）。

当通过本地 APIC 发送和接收一个 0 到 15 之间的中断向量时，APIC 会在其错误状态寄存器中指出有一个非法向量（参看 8.5.3.“错误处理”）。IA-32 架构保留 16 到 31 之间的向量作为预定义的中断、异常和 Intel 保留的编码（参看表 5-1）；然而，本地 APIC 不会把这些向量当作非法的。

当一个非法向量值（0 到 15）被写进一个 LVT 项，且传送模式是固定的时，APIC

可能会发出一个非法向量错误的信号，而不管屏蔽位是否置位，或者事实上是否看见一个中断输入。

8.5.3.错误处理

本地 APIC 提供一个错误状态寄存器（ESR），用来在处理中断的过程中探测到错误时记录错误（参看图 8-9）。当本地 APIC 在 ESR 中设置一个错误位时，就产生一个 APIC 错误中断。当探测到一个错误时，LVT 错误寄存器允许选择一个中断向量传送给处理器核心。LVT 错误寄存器也提供屏蔽 APIC 错误中断的方法。

ESR 标志的功能如下：

发送校验和错误 （仅对 P6 系列和 Pentium 处理器。）当本地 APIC 探测到它发送在 APIC 总线上的消息有校验和错误时，则置位。

接收校验和错误 （仅对 P6 系列和 Pentium 处理器。）当本地 APIC 探测到它从 APIC 总线上接收的消息有校验和错误时，则置位。

发送接受错误 （仅对 P6 系列和 Pentium 处理器。）当本地 APIC 探测到它发送的消息没有被 APIC 总线上的任何 APIC 接受时，则置位。

接收接受错误 （仅对 P6 系列和 Pentium 处理器。）当本地 APIC 探测到它接到的消息没有被 APIC 总线上的包括它自己在内的任何 APIC 接受时，则置位。

发送非法向量 当本地 APIC 探测到它发送的消息中有一个非法向量时，则置位。

接收非法向量 当本地 APIC 探测到它接收到的消息中有一个非法向量时，包括本地向量表和一个自我中断中的非法向量代码，则置位。

非法向量寄存器地址 （仅对 Pentium 4、Intel Xeon、P6 系列处理器。）当处理器试图访问本处理器的本地 APIC 寄存器地址空间中的一个未实现的寄存器时，则置位。也就是在 APIC 寄存器基地址（在 IA32_APIC_BASE MSR 中指定的）加上 4K 字节的范围内。

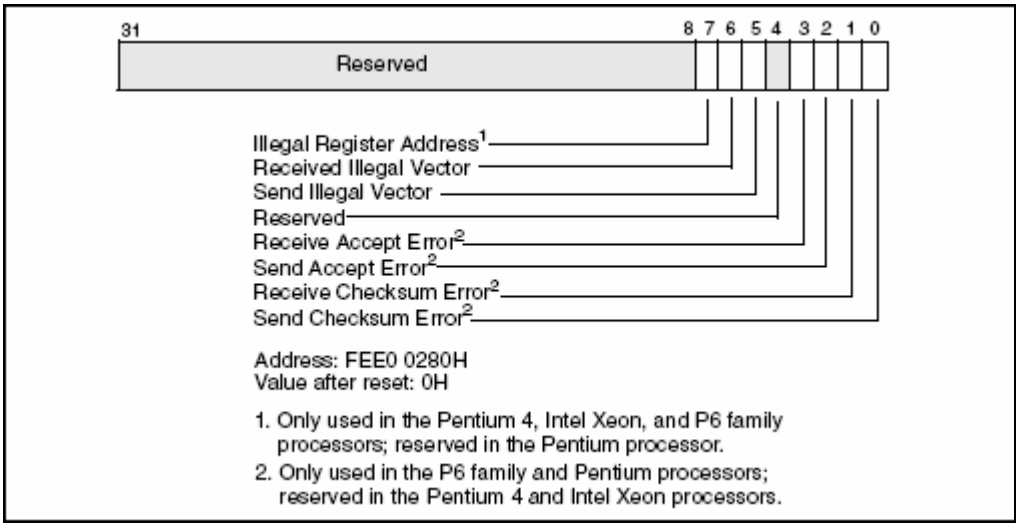

图 8-9 错误状态寄存器（ESR）

ESR 是一个读写寄存器。更新 ESR 寄存器时，写任何一个值到 ESR 必须在读 ESR 之前进行。第一次写将导致 ESR 的内容被更新为最近的错误状态。一步一步后退（back-to-back）写清除 ESR 寄存器。

寄存器中的一个错误位被置位后，它会一直保持到寄存器被清除为止。设置 LVT 错误寄存器的屏蔽位，会阻止错误被记录到 ESR 中；但是，在屏蔽位被设置之前的 ESR 的状态将得到保持。

8.5.4.APIC 计时器

本地 APIC 单元包含一个 32 位可编程计时器，供软件进行事件或者操作计时之用。设置这个计时器需要对四个寄存器进行编程：划分配置寄存器（图 8-10）、初始计数和当前计数寄存器（图 8-11）、LVT 计时器寄存器（图 8-8）。

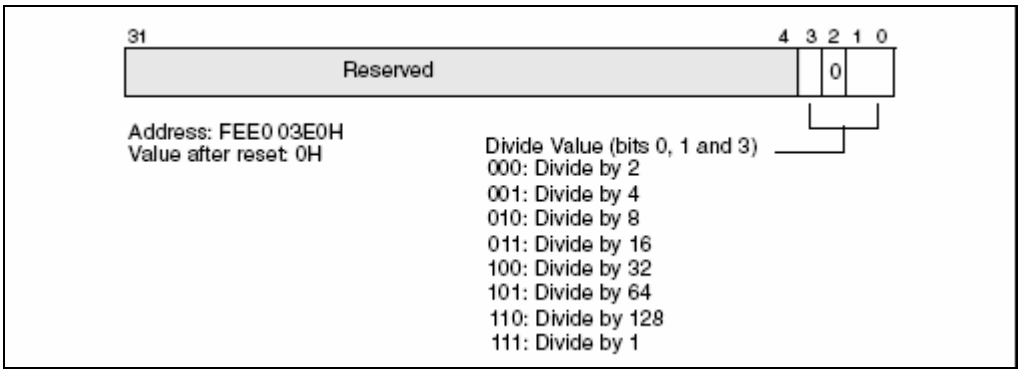

图 8-10 划分配置寄存器

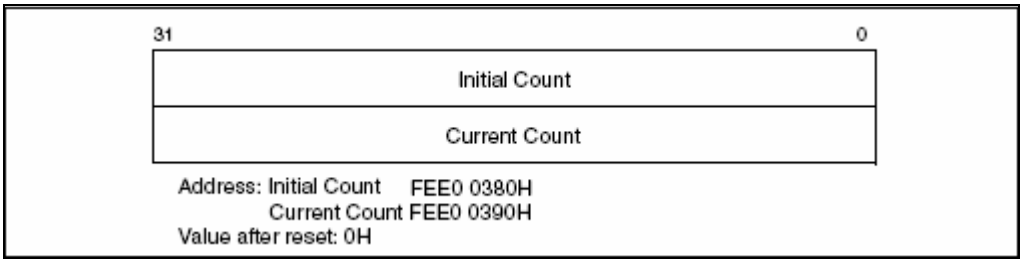

图 8-11 初始计数和当前计数寄存器

计时器的时间基准，来自于处理器的总线时钟除以划分配置寄存器中的值所得的值。

通过计时器的 LVT 项可以把计时器的操作模式配置成一次性的或者周期性的。在一次性操作模式中，通过对初始计数寄存器编程来启动计时器。然后，初始计数值就被复制到当前计数寄存器中，并开始倒计数。当计数器到达 0 时，就产生一个计数器中断，并保持这个 0 值，直到再次编程。

在周期性操作模式中，一旦计数达到 0，当前计数寄存器就自动从初始计数寄存器中重新加载计数值，产生一个计数器中断，继续开始倒计数。如果倒计数过程中初始计数寄存器被设置了，则将使用新的初始计数值，重新开始计数。初始计数寄存器是一个读——写寄存器；当前计数寄存器是只读的。

由 LVT 计时器寄存器决定，当计时器计数到达 0 时，随着计时器中断传送给处理器的向量号。LVT 计时器寄存器中的屏蔽位可用来屏蔽计时器中断。

8.5.5.本地中断接受

当一个本地中断发送到处理器核心时，它就要遵从图 8-17 中的中断接受流程图中的接受准则。如果中断被接受了，则把它记录到 IRR 寄存器中，处理器按照它的优先级来进行处理（8.8.4.“固定中断的中断接受”）。如果中断未被接受，则把它发回给本地 APIC，并重试。

8.6.发出处理器间中断

下面各节描述用来供软件发送处理器间中断的本地 APIC 设施。用来发送 IPI 的主要本地 APIC 设施是中断命令寄存器（ICR）。ICR 可用于下列功能：

- 发送一个中断给另外一个处理器。
- 允许处理器转发它收到的一个中断，但不对另一个处理器的请求提供服务。
- 把处理器定向到中断本身（执行一次自我中断）。
- 传送特定的 IPI，比如启动 IPI（SIPI）消息，到其它处理器。

系统中，用这个设施产生的中断是通过系统总线（Pentium 4 和 Intel Xeon 处理器）或者 APIC 总线（P6 系列和 Pentium 处理器）传送到其它处理器的。处理器发送最低优先权的 IPI 的能力是与模型相关的，BIOS 和操作系统软件应该避免它。

8.6.1.中断命令寄存器（ICR）

中断命令寄存器（ICR）是一个 64 位本地 APIC 寄存器（参看图 8-12），允许运行在处理器上的软件指定和发送处理器间中断（IPI）给系统中的其它 IA-32 处理器。

要发送一个 IPI，软件必须设置 ICR 以指明将要发送的 IPI 消息的类型和目的处理器或处理器组。（除了传送状态域是只读的之外，ICR 的所有其它域都是可读——写的。）写 ICR 的低位双字的操作，将导致 IPI 发送。

ICR 由下面的域构成：

向量 发送的中断向量号

传送模式 指定将要发送 IPI 类型。这个域也叫做 IPI 消息类型域。

000（固定的） 传送向量域中指定的中断到目标处理器或者处理器组。

001(最低优先权) 除了是把中断传送给目的域中指定的目标处理器组中的优先级最低的处理器之外，其它同于固定模式。处理器传送最低优先权 IPI 的能力是与模型相关的，BIOS 和操作系统软件应该避免它。

010（SMI） 传送一个 SMI 中断给目标处理器或者处理器组。为了与未来兼容，该向量域应该设成 00H。

011（保留）

100（NMI） 传送一个 NMI 中断给目标处理器或者处理器组。忽略向量信息。

101（INIT） 传送一个 INIT 请求给目标处理器或者处理器组，使之执行一次初始化。作为这个 IPI 消息的结果之一，所有处理器都执行

一次初始化。为了与未来兼容，这个向量域应该设成 00H。

101（未激活 INIT 的电平） （Pentium 4 和 Intel Xeon 处理器中不支持。）发送一个同步消息给系统中的所有本地 APIC，把它们的仲裁 ID（存储在它们的 Arb ID 寄存器中）设置成它们的 APIC ID（参看 8.7.“系统和 APIC 总线仲裁”）。对于这个传送模式，电平标志必须设为 0，触发模式标志设为 1。不管目的域或者目的简略域的值是什么，这个 IPI 要发给所有处理器；然而，软件应该指定“包括自己的所有”这种简略形式。

110（启动） 发送一个特殊的“启动”IPI（叫做 SIPI）给目标处理器或者处理器组。典型情况下，这个向量指向一个启动例程，是 BIOS 自举代码的组成部分（参看 7.5.“多处理器（MP）初始化”）。注意，用这种传送模式发送的 IPI 不会自动重试，如果源 APIC 不能传送它的话。这个该由软件来确定 SIPI 是否成功传送，并在必要时重新发送。

目的模式 选择物理的（0）或者逻辑的（0）目的模式（参看 8.6.2.“确定 IPI 目的”）。

传送状态（只读） 指明 IPI 的传送状态，具体如下：

0（空闲） 该本地 APIC 当前没有 IPI 活动，或者该本地 APIC 的前一个 IPI 已被发送，并且被目标处理器或者处理器组接受。

1（待发送） 指明该本地 APIC 发送的最后一个 IPI 还未被目标处理器或者处理器组接受。

电平 对于未激活电平的 INIT 传送模式来说，这个标志必须被置为 0；对于所有其它传送模式来说，必须被置为 1。（Pentium 4 和 Intel Xeon 处理器中，这个标志没有意义，总是作为 1 发出。）

触发模式 当使用 INIT 的未激活电平传送模式时，选择触发模式：边极的（0）或电平的（1）。所有其它传送模式忽略它。（Pentium 4 和 Intel Xeon 处理器中，这个标志没有意义，总是作为 0 发出。）

目的简略 指出是否使用简略符号来指定中断的目的，如果是的话，那么，使用了哪种简略形式。目的简略用来代替 8 位目的域，软件单写一次到 ICR 双字的低字

节即可发送。简略为下列这些情况定义的：软件自我中断、给系统中包括发送者在内的所有处理器的 IPI、给系统中发送者除外的所有处理器的 IPI。

00（无简略） 目的在目的域中定义。

01（自我） 发送 APIC 也是 IPI 的唯一目的（APIC）。这个目的简略允许软件中断它正在运行的处理器。一个 APIC 实现在内部随意发送自我中断消息，或者发送消息给总线，同时把它当作其它 IPI 来“探测”它。

10（包括自我的所有） 这个 IPI 发送给系统中包括发送这个 IPI 的处理器在内的所有处理器。APIC 将广播一个目的域设为 FH 的 IPI 消息给 Pentium 和 P6 系列处理器，和目的域设为 FFH 的给 Pentium 4 和 Intel Xeon 处理器。

11（除去自我的所有） 这个 IPI 发送给系统中除了发送这个 IPI 的处理器之外的所有处理器。APIC 将广播一个带有物理目的模式并且目的域设为 0xFH 的 IPI 消息给 Pentium 和 P6 系列处理器，和目的域设为 0xFFH 的给 Pentium 4 和 Intel Xeon 处理器。这个目的简略连同最低优先权传送模式的支持是与模型相关的。对于 Pentium 4 和 Intel Xeon 处理器来说，当这个目的简略连同最低优先权传送模式一起使用时，则 IPI 可能会重新定向回那个发送处理器。

目的 指定目标处理器或者处理器组。仅当目的简略域设成 00B 时，才使用这个域。如果目的模式被设成物理的，那么，对于 Pentium 和 P6 系列处理器来说，第 56 到 59 位包含目标处理器的 APIC ID；对于 Pentium 4 和 Intel Xeon 处理器来说，第 56 到 63 位包含目标处理器的 APIC ID。如果目的模式被设成逻辑的，那么，8 位目的域的解释，取决于系统中所有处理器的本地 APIC 的 DFR 和 LDR 寄存器的设置（参看 8.6.2.“确定 IPI 目的”）。

注意，不是 ICR 的所有选项组合都是合法的。表 8-2 显示了 Pentium 4 和 Intel Xeon 处理器中，ICR 中域的合法组合；表 8-3 显示了 P6 系列处理器中，ICR 中域的合法组合。

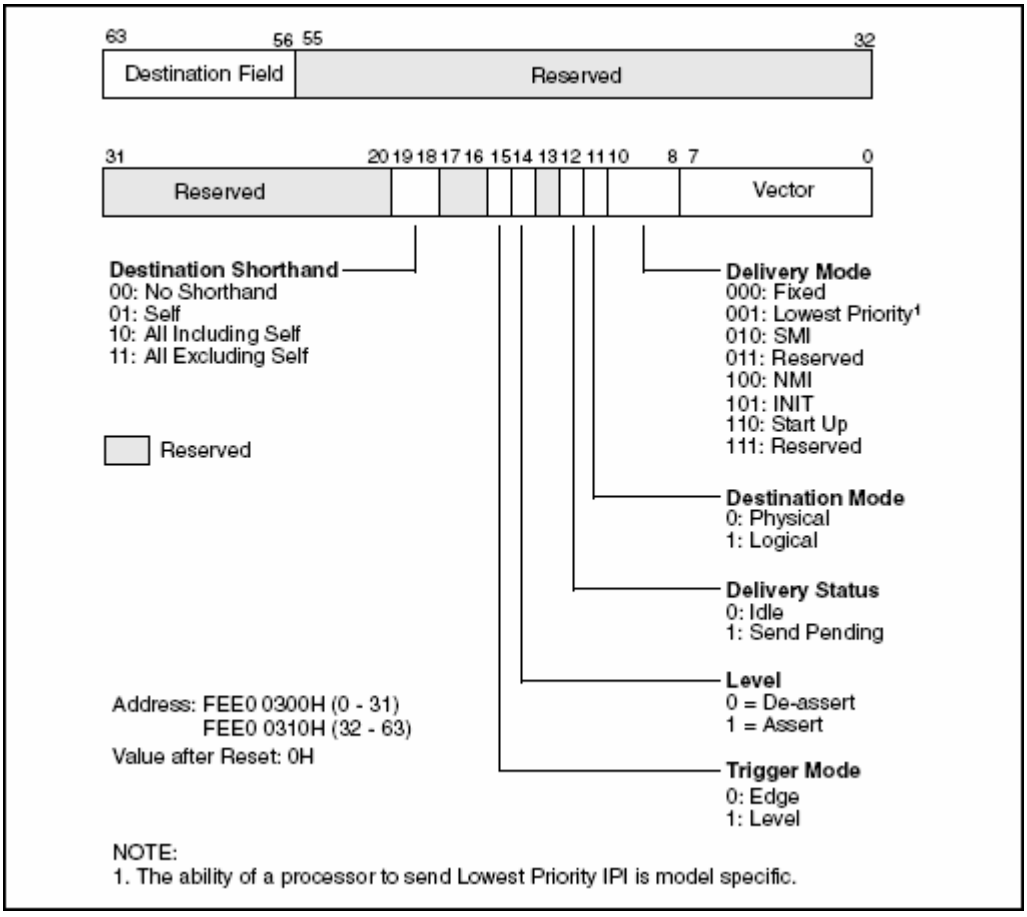

图 8-12 中断命令寄存器（ICR）

表 8-2 Pentium 4 和 Intel Xeon 处理器中本地 APIC 的 ICR 的合法组合

目的简略	合法/非法	触发模式	传送模式	目的模式
没有简略	合法	边极	所有模式 [1]	物理或逻辑
没有简略	非法 [2]	电平	所有模式	物理或逻辑
自我	合法	边极	固定模式	X [3]
自我	非法 [2]	电平	固定模式	X
自我	非法	X	最低优先权、NMI、INIT、SMI、启动	X
包括自我的所有	合法	边极	固定模式	X
包括自我的所有	非法 [2]	电平	固定模式	X
包括自我的所有	非法	X	最低优先权、NMI、INIT、SMI、启动	X
除去自我的所有	合法	边极	固定、最低优先权 [1, 4]、NMI、INIT、SMI、启动	X
除去自我的所有	非法 [2]	电平	固定、最低优先权 [4]、NMI、INIT、SMI、启动	X

注：

1．处理器发送最低优先权的 IPI 的能力是与模型相关的。

2．对于这些中断，如果触发模式位是 1（电平），则本地 APIC 将覆盖这个位的设置，并且把这个中断做为边极中断发出。

3．X——忽略该设置。

4．当使用“最低优先权”传送模式和“除去自我的所有”目的时，IPI 可被重定向回发出的 APIC，事实上它与“包括自我的所有”目的模式一样。

表 8-3 P6 系列处理器中本地 APIC 的 ICR 的合法组合

目的简略	合法/非法	触发模式	传送模式	目的模式
没有简略	合法	边极	所有模式[1]	物理或逻辑
没有简略	合法[2]	电平	固定、最低优先权[1]、NMI	物理或逻辑
没有简略	合法[3]	电平	INIT	物理或逻辑
自我	合法	边极	固定模式	X[4]
自我	1	电平	固定模式	X
自我	非法[5]	X	最低优先权、NMI、INIT、SMI、启动	X
包括自我的所有	合法	边极	固定模式	X
包括自我的所有	合法[2]	电平	固定模式	X
包括自我的所有	非法[5]	X	最低优先权、NMI、INIT、SMI、启动	X
除去自我的所有	合法	边极	所有模式[1]	X
除去自我的所有	合法[2]	电平	固定、最低优先权[1]、NMI	X
除去自我的所有	非法[5]	电平	SMI、启动	X
除去自我的所有	合法[3]	电平	INIT	X
X	非法[5]	电平	SMI、启动	X

注：

1．处理器发送最低优先权的 IPI 的能力是与模型相关的。

2．如果电平位置 1，则当作边极触发。其它忽略。

3．当电平位置 1 时，则当作边极触发；当电平位置 0（未激活）时，则当作“INIT 电平未激活”消息。只有 INIT 电平未激活消息允许设置这个电平位为 0。对于所有其它消息，这个电平位必须置为 1。

4．X——不关注。

5．APIC 的行为未定义。

8.6.2.确定 IPI 目的

一个 IPI 的目的可以是系统总线上的一个、所有或者一组处理器。IPI 发送者使用下列 APIC 寄存器或者寄存器中的域，来指定 IPI 的目的：

- ICR 寄存器——ICR 寄存器中的下列域用来指定一个 IPI 的目的：
  - 目的模式——选择两个模式中的一个（物理的或者逻辑的）。
  - 目的域——在物理目的模式，用来指定目的处理器的 APIC ID；在逻辑处

理器中，用来指定一个消息目的地址（MDA），可用来选择集群中的特定处理器。

— 目的简略——一个指定所有处理器、除去自我的所有或者自我做为目的的快捷方法。

— 传送模式，最低优先权——从架构上指定一个最低优先权的仲裁机制，用来从一组特定的处理器中选择一个目的处理器的。处理器发送最低优先权 IPI 的能力是与模型相关的，BIOS 和操作系统软件应该避免它。

- 本地目的寄存器（LDR）——与逻辑目的模式和 MDA 一起使用，用来选择目的处理器。
- 目的格式寄存器（DFR）——与逻辑目的模式和 MDA 一起使用，用来选择目的处理器。

如何使用 ICR、LDR 和 DFR 来选择一个 IPI 目的，取决于使用的目的模式：物理的、逻辑的、广播/自我、或者最低优先权传送模式。这些目的模式将在后续各节描述。

8.6.2.1.物理目的模式

在物理目的模式下，目的处理器是由它的本地 APIC ID 指定的（参看 8.4.6.“本地 APIC ID”）。对于 Pentium 4 和 Intel Xeon 处理器来说，一个单个目的（本地 APIC ID 从 00H 到 FEH）或对所有 APIC 的广播（APIC ID 是 FFH）都可能在物理模式下指定。

物理目的模式不支持一个广播 IPI（MDA 的第 28 到 31 位都是 1）或者具有最低优先权传送模式的 I/O 子系统的初始中断，不必用软件配置它们。对于非广播的 IPI 或者具有最低优先权传送模式的 I/O 子系统的初始中断，软件必须确保定义在中断地址中的 APIC 是存在的，并且是开启的，能接收中断。

对于 P6 系列和 Pentium 处理器来说，一个单个目的是由具有从 0H 到 0EH 的本地 APIC ID 的物理目的传送模式指定的，允许 APIC 总线中访问多达 15 个本地 APIC。对所有本地 APIC 的广播是由 0FH 指定的。

注意

系统总线中可以访问的本地 APIC 的实际数量是由硬件限制的。

8.6.2.2.逻辑目的模式

在逻辑目的模式下，IPI 目的是用 8 位消息目的地址（MDA）来指定的，输入在 ICR

的目的域中。一旦收到一个使用逻辑目的模式发送的 IPI 消息，则本地 APIC 使用它的 LDR 和 DFR 中的值与消息中的 MDA 比较，以确定它是否应该接受和处理这个 IPI。对于逻辑目的模式的两种配置来说，当与最低优先权的传送模式结合起来时，则由软件负责确保，IPI 或者 I/O 子系统中断中包含的或者访问到的本地 APIC 存在，并且是开启的，能接收中断。

图 8-13 展示了逻辑目的寄存器（LDR）的格式。这个寄存器中的 8 位逻辑 APIC ID 域是用来创建能与 MDA 比较的标识符的。

注意

逻辑 APIC ID 不应该与本地 APIC ID 混淆，本地 APIC ID 包含在本地 APIC ID 寄存器中。

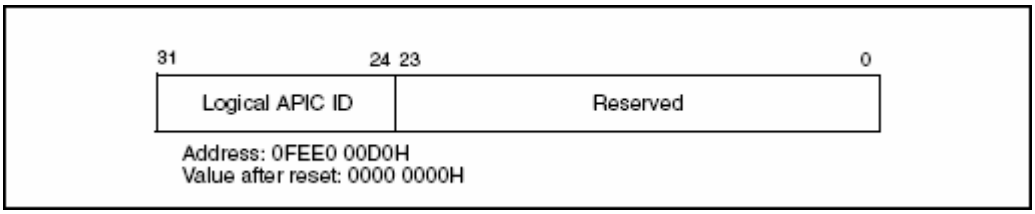

图 8-13 逻辑目的寄存器（LDR）

图 8-14 展示了目的格式寄存器（DFR）的格式。这个寄存器中的 4 位模型域选择两个模型（平坦或者集群）中的一种，可用来在使用逻辑目的模式时解释 MDA。

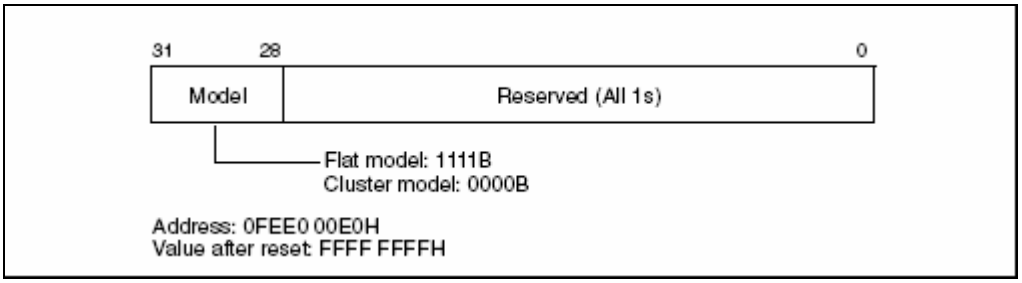

图 8-14 目的格式寄存器（DFR）

两个模型的 MDA 的解释在下列段落描述。

平坦模型。编程把 DFR 的第 28 到 31 位设为 1111，即可选择这个模型。此时，通过为每个本地 APIC 的 LDR 的逻辑 APIC ID 设置不同的位，可以为多达 8 个逻辑 APIC 建立唯一的逻辑 APIC ID。然后，通过设置 MDA 中的一位或者几个位就可以选择一组逻辑 APIC。

每一个逻辑 APIC 对 MDA 和它的逻辑 APIC ID 执行位的与操作。如果探测到一个真条件，则本地 APIC 接受 IPI 消息。通过把 MDA 设置成全 1，可以获得一个对所有 APIC

的广播。

集群模型。编程把 DFR 的第 28 到 31 设为 0000，即可选择此模型。该模型支持两个基本的目的方案：平坦集群和层次集群。

只有 P6 系列和 Pentium 处理器支持平坦集群目的模型。使用此模型时，假定所有 APIC 都通过 APIC 总线连接的。MDA 的第 28 到 31 位包含目的集群的编码地址，第 24 到 27 位标识集群内的四个本地 APIC（跟平坦连接模型一样，每个位指派给一个本地 APIC）。为了标识一个或者多个本地 APIC，用 MDA 的第 28 到 31 位与 LDR 的第 28 到 31 位进行比较，以确定某个本地 APIC 是否是本集群的一部分。用 MDA 的第 24 到 27 位与 LDR 的第 24 到 27 位进行比较，以标识某个本地 APIC 是否在本集群中。

通过写 MDA 的目标集群地址中的第 28 到 31 位，并设置选择的第 24 到 27 位（相应于集群的被选成员），可以指定集群中的处理器组。在这种模式下，15 个集群中的每一个(集群地址为 0 到 14)都可以在消息中指定 4 个本地 APIC。但是，对于 P6 和 Pentium 处理器的本地 APIC 来说，APIC 仲裁 ID 仅支持 15 个代理，因而，这种模式中支持的处理器和它们的本地 APIC 的总数限制为 15 个。把所有的目的位都设为 1 可以得到向所有本地 APIC 的广播。这可以保证所有集群都匹配，并在每个集群中选择所有本地 APIC。集群模式不支持广播 IPI 或者最低优先权传送模式的 I/O 子系统广播中断，也不必由软件配置。

Pentium 4、Intel Xeon、P6 系列或者 Pentium 处理器都可以使用层次集群目的模型。使用这个模型，经由独立系统或者 APIC 总线连接不同的平台集群，可以建立一个层次网络。这个方案需要每个集群中有一个集群管理者，由它负责在系统或者 APIC 总线间处理消息传递。一个集群包含四个代理。因此，15 个集群管理者，每个有四个代理，可以组成一个多达 60 个 APIC 代理的网络。注意，层次 APIC 网络需要一个特殊的集群管理设备，它不是本地或者 I/OAPIC 单元的组成部分。

8.6.2.3.广播/自我传送模式

在广播 IPI 给总线上的所有处理器并且/或者发回它自己的时候，ICR 的目的简略域允许绕过这种传送模式（参看 8.6.1.“中断命令寄存器（ICR）”）。支持三个目的简略：自我、除去自我的所有、包括自我的所有。当使用一个目的简略时，忽略目的模式。

8.6.2.4.最低优先权传送模式

在最低优先权模式下，可以对 ICR 编程使之发送一个 IPI 给系统总线上的几个处理器，使用逻辑或者简略目的机制选择这些处理器。被选的处理器就在系统总线或者 APIC 总线上相互仲裁，让最低优先权的处理器接受这个 IPI。

对于基于 Intel Xeon 处理器的系统来说，芯片组总线控制器从系统中的 I/O APIC 代理接受消息，并把中断定向到系统总线上的处理器上。当使用最低优先权传送模式时，芯片组在可能的目标组之外选择一个目标处理器去接收中断。Pentium 4 处理器在系统总线上提供一个特殊的总线周期，把系统中的每个逻辑处理器的当前任务优先级通知给芯片组。芯片组保存这个信息，并在接收中断时用它选择最低优先权处理器。

对于基于 P6 系列处理器的系统来说，用于最低优先权仲裁的处理器优先级保存在每个本地 APIC 的仲裁优先级寄存器（APR）中。图 8-15 展示了 APR 的结构。

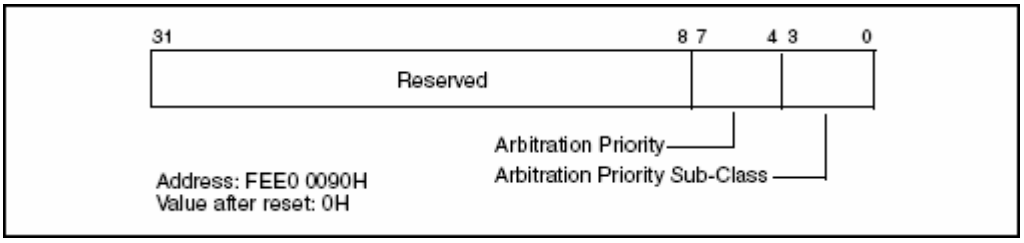

图 8-15 仲裁优先级寄存器（APR）

APR 值按下列方式计算：

```
IF (TPR[7:4] ≥ IRRV[7:4]) AND (TPR[7:4] > ISRV[7:4])
   THEN
      APR[7:4] ← TPR[7:0]
   ELSE
      APR[7:4] ← max(TPR[7:4] AND ISRV[7:4], IRRV[7:4])
      APR[3:0] ← 0.
```

其中，TPR 值是 TPR 中的任务优先级值（参看图 8-18），IRRV 值是设置在 IRR（参看图 8-20）中的最高优先权位的向量号，或者是 00H（如果 IRR 位没有设置的话）；ISRV 值是设置在 ISR（参看图 8-20）中的最高优先权位的向量号。按照目的处理器中的仲裁，APR 值最小的处理器处理 IPI，其它处理器忽略它。

（P6 系列和 Pentium 处理器。）对于这些处理器，如果**焦点（focus）处理器**存在，它可能接受中断，而不管它的优先级。如果一个处理器当前正在服务中断或者有那个

中断的待处理请求，则把这个处理器叫做焦点。对于 Intel Xeon 处理器来说，不支持焦点处理器的概念。

在使用最低优先权传送模式但不更新 TPR 的操作系统中，保存在芯片组中的 TPR 信息，总是导致中断从逻辑组传送到同样的处理器。这个行为从功能上来说是对 P6 系列处理器向后兼容的，但是，可能会导致不可预料的性能方面的牵连。

8.6.3. IPI 传送和接受

当向 ICR 低半截的双字写入时,本地 APIC 就根据包含在 ICR 中的信息创建一个 IPI 消息，并发送到系统总线上（Pentium 4 和 Intel Xeon 处理器）或者 APIC 总线上（P6 系列和 Pentium 处理器)。发送之后，这些 IPI 被处理的方式在 8.8.“处理中断”中描述。

8.7. 系统和 APIC 总线仲裁

当几个本地 APIC 和 I/O APIC 正在系统总线（或 APIC 总线）上发送 IPI 和中断消息时，发送和处理消息的顺序通过总线仲裁来决定。

对于 Pentium 4 和 Intel Xeon 处理器来说，本地 APIC 和 I/O APIC 使用为总线定义的仲裁机制来决定 IPI 被处理的顺序。这种机制不是架构上的，也不能由软件控制。

对于 P6 系列和 Pentium 处理器来说，本地 APIC 和 I/O APIC 使用基于 APIC 的仲裁机制来决定 IPI 被处理的顺序。这里，每个本地 APIC 被赋予一个从 0 到 15 的仲裁优先级，仲裁期间，I/O APIC 使用它来确定应该允许哪个本地 APIC 访问 APIC 总线。具有最高仲裁优先级的本地 APIC 获得总线访问。一旦完成一个仲裁回合，则获得访问权限的本地 APIC 就把它的仲裁优先级降为 0,而未获得访问权的所有本地 APIC 都把它们的权限升一个 1。

本地 APIC 的当前仲裁优先级保存在 4 位，对软件透明的仲裁 ID（Arb ID）寄存器中。在复位期间，这个寄存器被初始化成该 APIC ID 号（保存在本地 APIC ID 寄存器中)。同时发出的未激活电平的 INIT 的 IPI，以及 ICR 命令，通过把 Arb ID 寄存器的每个代理复位成它的当前 APIC ID 的值，可用来重新同步本地 APIC 的仲裁优先级。（Pentium 4 和 Intel Xeon 处理器没有实施 Arb ID 寄存器。）

8.10.“APIC 总线消息传递机制和协议（P6 系列和 Pentium 处理器)”，描述 APIC

总线仲裁协议和总线消息格式；而 8.6.1.“中断命令寄存器（ICR）”，描述未激活电平的 INIT 的 IPI 消息。

注意，除了 SIPI 这个 IPI 之外（参看 8.6.1.“中断命令寄存器（ICR）”），所有不能成功地传送到它们目的的总线消息，会自动重试。软件应该避免这种状况，即发送 IPI 到关闭的或者不存在的本地 APIC，进而导致消息重复再发送。

8.8.处理中断

当一个本地 APIC 收到一个来自本地源一个中断，来自一个 I/O APIC 的一个中断或 IPI 时，它处理消息的方式取决于处理器实现，具体描述见下面各节。

8.8.1.Pentium 4 和 Intel Xeon 处理器的中断处理

在 Pentium 4 和 Intel Xeon 处理器中，本地 APIC 处理它接受到的本地中断、中断消息和 IPI 的步骤如下：

1．确定是否是指定目的（参看图 8-16）。如果是，就接受消息；如果不是，则丢弃消息。

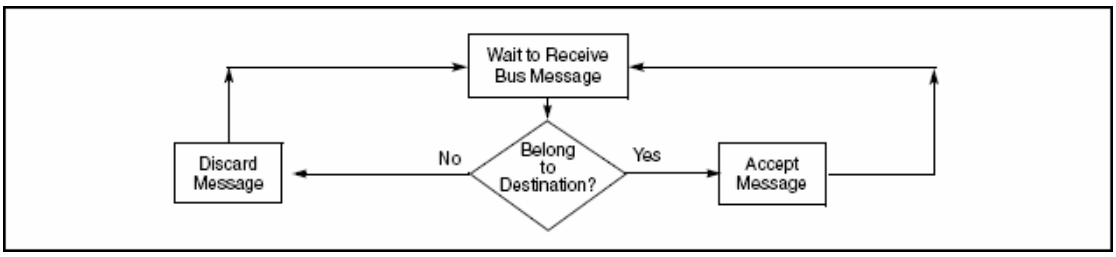

图 8-16 本地 APIC（Pentium 4 和 Intel Xeon 处理器）的中断接受流程图

2.如果本地 APIC 确定是该中断指派的目的，并且中断请求是一个 NMI、SMI、INIT、ExtINT 或者 SIPI，则把该中断直接发送到处理器核心处理。

3．如果本地 APIC 确定是该中断指派的目的，但是中断请求不是第 2 步所列中断中的某一个，则本地 APIC 在 IRR 中设置相应的位。

4．当中断悬挂在 IRR 和 ISR 寄存器中时，本地 APIC 根据它们的优先级以及 TPR 和 PPR 中的当前任务和处理器优先级，一次一个地把它们调度给处理器（参看 8.8.3.1.“任务和处理器优先级”）。

5．当一个固定中断已经被调度给处理器核心请求处理时，该处理例程的完成是由

处理例程代码写本地 APIC 的中断结束（EOI）寄存器来标示的（8.8.5.“发中断服务完成信号”）。写 EOI 寄存器操作导致本地 APIC 从 ISR 队列中删除该中断，（对于电平触发中断）并在总线上发送一个消息说明中断处理已经结束。（对 EOI 寄存器的一个写操作不是必须包含在一个 NMI、SMI、INIT、ExtINT 或者 SIPI 的处理例程中。）

8.8.2.P6 系列和 Pentium 处理器的中断处理

在 P6 系列和 Pentium 处理器中，本地 APIC 处理它接受到的本地中断、中断消息和 IPI 的步骤如下（参看图 8-17）：

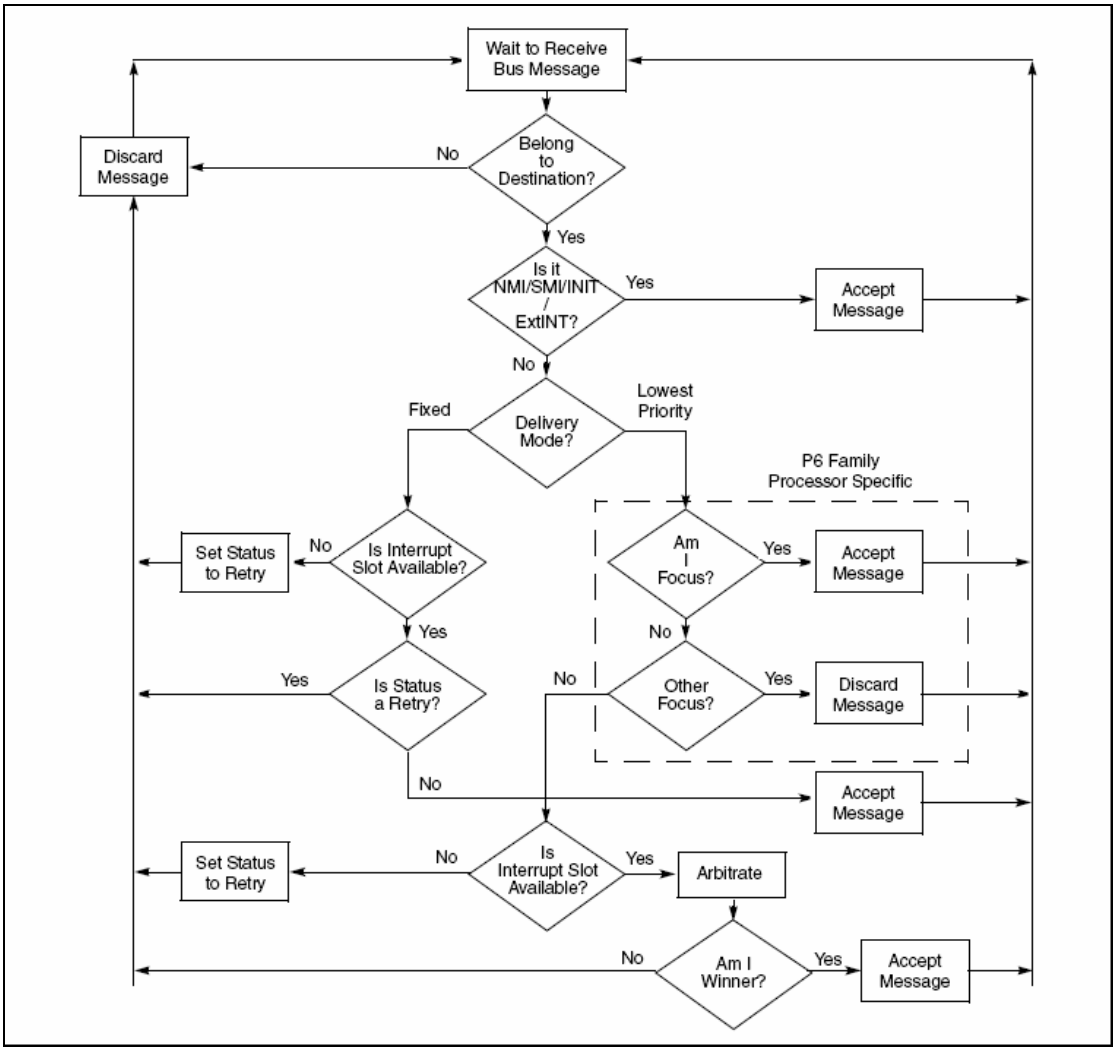

图 8-17 本地 APIC（P6 系列和 Pentium 处理器）的中断接受流程图

1.（仅对 IPI。）检查 IPI 消息以确定它是否是该 IPI 消息的指定目的，具体描述参见 8.6.2.“确定 IPI 目的”。如果是指定目的，就继续它的接受过程；如果不是目的，就丢弃该 IPI 消息。当该消息指定最低优先权传送模式时，本地 APIC 将和该 IPI 消息

的接收者中指派的其它处理器一起仲裁。

2. 如果本地 APIC 确定它是该中断的指派目的，并且该中断请求是 NMI、SMI、INIT、ExtINT、未激活的 INIT 中断、或者 MP 协议 IPI 消息之一（BIPI、FIPI、SIPI），则把该中断直接发往处理器核心请求处理。

3. 如果本地 APIC 确定是该中断指派的目的，但是中断请求不是第 2 步所列中断中的某一个，则本地 APIC 在 IRR 和 ISR 寄存器包含的两个待处理中断队列之一寻找一个空槽（参看图 8-20）。如果有一个槽可用（参看 8.8.4“固定中断的中断接受”），则把该中断放置在该槽中。如果没有槽可用，它就拒绝该中断请求，并附带一个重试消息把它发回发送者。

4. 当中断悬挂在 IRR 和 ISR 寄存器中时，本地 APIC 根据它们的优先级以及 TPR 和 PPR 中的当前任务和处理器优先级，一次一个地把它们调度给处理器（参看 8.8.3.1.“任务和处理器优先级”）。

5. 当一个固定中断已经被调度给处理器核心请求处理时，该处理例程的完成是由处理例程代码写本地 APIC 的中断结束（EOI）寄存器来标示的（8.8.5.“发中断服务完成信号”）。写 EOI 寄存器操作导致本地 APIC 从它的队列中删除该中断，（对于电平触发中断）并在总线上发送一个消息说明中断处理已经结束。（对 EOI 寄存器的一个写操作不是必须包含在一个 NMI、SMI、INIT、ExtINT 或者 SIPI 的处理例程中。）

后续各节将更加详细地描述本地 APIC 和处理器的中断的接受及其处理。

8.8.3. 中断、任务和处理器优先级

对于通过本地 APIC 传送给处理器的中断来说，每个中断隐含一个基于向量号的优先级。本地 APIC 使用这个优先级来决定什么时候服务该中断，相对于处理器的其余活动包括服务其它中断来说。

对于范围在 16 到 255 的中断向量来说，中断优先级使用下列关系式来确定：

优先级 = 向量/16

这里，商下舍入到最近的整数值来确定优先级，1 是最低优先权，15 是最高优先权。因为第 0 到 31 号向量是保留给 IA-32 架构专门使用的，因此，用户定义的中断优先级范围是 2 到 15。

每一个中断优先级（有时被软件解释为中断优先类）包含 16 个向量。中断的优先

级的次序是由向量号来区分的。向量号越大，那个优先级的优先权越高。在决定一个向量的优先权及其在一个优先权组里的向量等级时，该向量号通常被分为两个部分，向量的高 4 位表明它的优先权，低 4 位表明它在优先权组里的等级。

8.8.3.1.任务和处理器优先权

本地 APIC 也定义一个任务优先权和一个处理器优先权，并使用它来决定中断被处理的顺序。任务优先权是一个软件选择的值，范围在 0 到 15 之间（参看图 8-18），写进任务优先权寄存器（TPR）。TPR 是一个读/写寄存器。

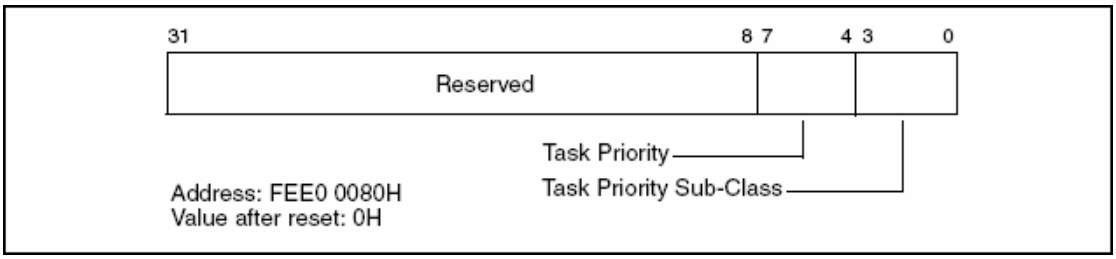

图 8-18 任务优先权寄存器（TPR）

注意

在这个讨论中，概念“任务”指的是一个软件定义的任务、进程、线程、程序或者例程，是被操作系统调度到处理器上运行的。并非指第 6 章“任务管理”中描述的 IA-32 架构定义的任务。

任务优先权允许软件去设置一个中断处理器的**优先权门限（threshold）**。处理器将只服务那些优先权限高于在 TPR 中指定的这个优先权门限的中断。如果软件设置 TPR 中的任务优先权为 0，则处理器将处理所有中断；如果设成 15，则除了那些随同 NMI、SMI、INIT、ExtINT、未激活的 INIT 和启动传送模式传送的中断之外，禁止所有其它中断的处理。这种机制使得操作系统能够暂时阻止特定中断干扰处理器正在进行的高优先权工作。

注意，任务优先权也用于确定本地处理器的仲裁优先权（参看 8.6.2.4.“最低优先权传送模式”）。

处理器优先权是由处理器设置的，值也是在 0 和 15 之间（参看图 8-19），写进处理器优先权寄存器（PPR）。PPR 是一个只读存储器。处理器优先权表示处理器正在运行的当前优先权。用来确定是否把一个待处理中断分发给处理器。

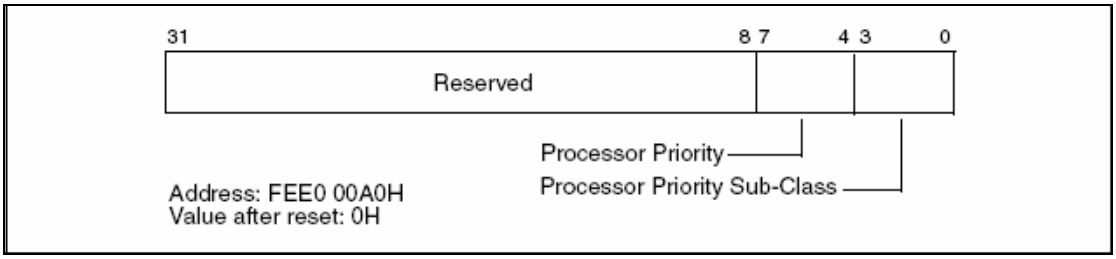

图 8-19 处理器优先权寄存器（PPR）

PPR 中的值可按下列方式计算：

```
IF TPR[7:4] ≥ ISRV[7:4]
   THEN
      PPR[7:0] ← TPR[7:0]
   ELSE
      PPR[7:4] ← ISRV[7:4]
      PPR[3:0] ← 0
```

这里，ISRV 值是最高优先权 ISR 位置位时的向量号，或者 ISR 位清除时则为 00H。事实上，处理器优先权或者设置为 ISR 中的最高优先权待处理中断，或者设置为当前任务优先权，实际结果取决于哪个权限更高。

8.8.4.固定中断的中断接受

本地 APIC 对从两个中断待处理寄存器（中断请求处理器（IRR）或在服务寄存器（ISR））之一中接受到的固定中断进行排队。这两个 256 位只读寄存器如图 8-20 所示。这两个寄存器中的 256 位代表 256 个可能的向量；0 到 15 号向量由 APIC 保留（参看 8.5.2.）。

注意

随同 NMI、SMI、INIT、ExtINT、启动或者未激活的 INIT 的传送模式的所有中断会绕过 IRR 寄存器和 ISR 寄存器，并被直接发送到处理器核心请求处理。

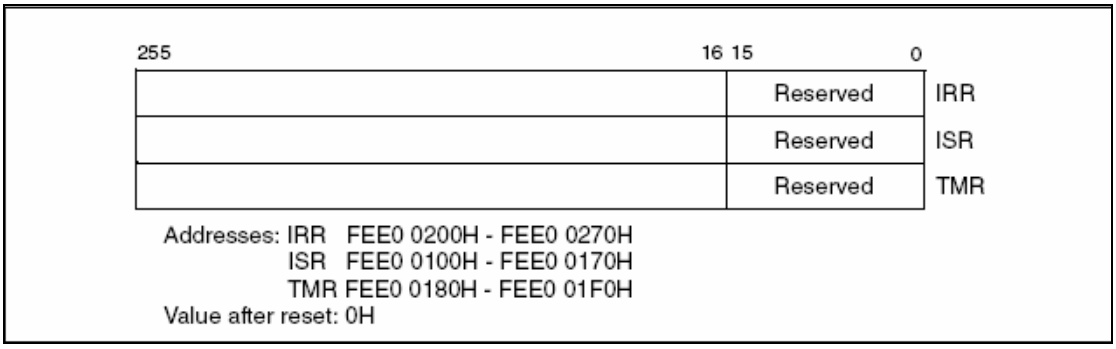

图 8-20 IRR、ISR 和 TMR 寄存器

IRR 包含接受到的活跃着的中断请求，但还没有调度给处理器请求服务。当本地 APIC 接受一个中断时，它把 IRR 中与接受的中断向量相对应的位置位。当处理器核心准备处理下一个中断时，本地 APIC 清除已经置位的最高优先权 IRR 位，并设置 ISR 相应的位。然后，把设置在 ISR 中的最高优先权位所对应的向量调度给处理器核心请求处理。

当处理器正在服务最高优先权中断时，通过设置 IRR 中的位，本地 APIC 可以发送额外的固定中断。当中断服务例程发出一个向 EOI 寄存器的写操作时（参看 8.8.5.“发中断服务完成信号”），本地 APIC 清除已设置的最高优先权 ISR 位作为响应。然后，它重复清除 IRR 中的最高权位和设置 ISR 中相应位的过程。然后，处理器核心开始执行 ISR 中设置的最高权位的服务例程。

如果同一个向量号有不止一个中断产生，则本地 APIC 会把向量在 IRR 和 ISR 中的位都设置。这就意味着，对于 Pentium 4 和 Intel Xeon 处理器来说，IRR 和 ISR 可以为每一个中断向量排两个中断：一个在 IRR 中，一个在 ISR 中。针对同一个中断向量发出的任何额外的中断都叠加在 IRR 中的单个位中。

对于 P6 系列和 Pentium 处理器来说，对每一个优先级，IRR 和 ISR 寄存器可以排队不超过两个中断，并且将拒绝同一优先级中收到的其它中断。

如果本地 APIC 收到一个优先权高于当前正在被服务的中断，并且这个中断对于处理器核心来说是可以接收的，则本地 APIC 立即把较高优先权的中断调度给处理器（无须等到一个对 EOI 寄存器的写操作）。然后，当前正在被执行的中断处理程序就被中断了，以便较高优先权的中断可被处理。一旦较高优先权中断完成，该中断被中断的服务将重新开始。

触发模式寄存器（TMR）指出中断的触发模式（参看 8-20）。一旦接受一个中断进入 IRR，针对边极触发的相应的 TMR 位就被清除，而针对电平触发的相应位就被置位。

当相应中断向量的 EOI 周期产生期间，如果一个 TMR 位是置位的，则 EOI 消息被发送给所有的 I/O APIC。

8.8.5.发中断服务完成信号

对于所有那些除了随同 NMI、SMI、INIT、ExtINT、启动或者未激活的 INIT 的传送模式之外的中断，中断处理程序必须包括一个对中断结束寄存器（EOI）的写（参看图 8-21）。这个写必须在中断处理例程结束处进行，有时在 IRET 指令之前。这个操作表明当前中断的服务完成了，并且本地 APIC 可以从 ISR 发出下一个中断了。

图 8-21 EOI 寄存器

一旦接收到一个 EOI，则 APIC 清除 ISR 中的最高级优先权位，并把下一个最高级优先权中断调度给处理器。如果结束的中断是一个电平触发中断，则本地 APIC 也发送一个中断结束消息给所有 I/O APIC。

为了与未来兼容，要求软件写一个 0 到 EOI 寄存器中来发出中断结束命令。

8.9.伪中断

当一个处理器升高它的任务优先权到大于或者等于处理器的 INTR 信号当前可被激活的那个中断级别时，就有可能会出现一种特殊状态。如果此时 INTA 周期发出了，将要分发的中断已经被屏蔽了（通过软件编程），则本地 APIC 将传送一个伪中断向量。分发伪中断向量不影响 ISR，因此，这个向量的处理程序的返回应该没有一个 EOI。

伪中断向量的向量号是在伪中断向量寄存器中指定的（参看 8-22）。该寄存器中的域功能如下：

伪 向量	当本地 APIC 产生一个伪向量时，决定传送给处理器的向量号。
	（对于 Pentium 4 和 Intel Xeon 处理器。）该域的第 0 到 7 位是由软件编程的。

	（对于 P6 系列和 Pentium 处理器。）该域的第 4 到 7 位是由软件编程的，第 0 到 3 位是由硬连线而成的逻辑位。软件对第 0 到 3 位的写是没有效果的。
APIC 软件开启/关闭	允许软件暂时开启（1）或者关闭（0）本地 APIC（参看 8.4.3.“开启或者关闭本地 APIC”）。
焦点处理器检验	当使用最低优先权传送模式时，确定是否开启（0）或者关闭（1）焦点处理器检验。在 Pentium 4 或者 Intel Xeon 处理器中，这个位是保留的，并且应该被清为 0。

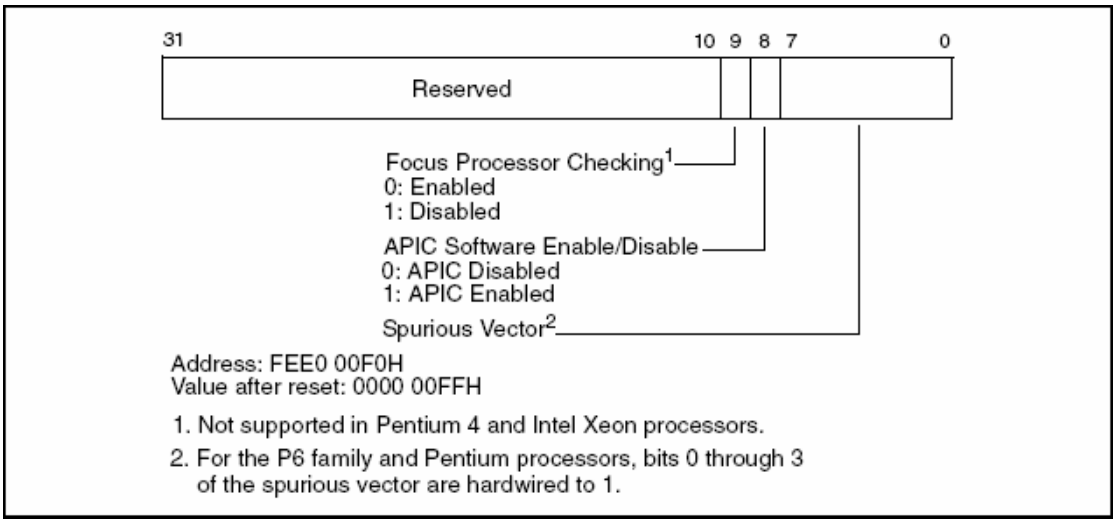

图 8-22 伪中断向量寄存器（SVR）

8.10.APIC 总线消息传送机制和协议（仅对 P6 系列和 Pentium 处理器）

Pentium 4 和 Intel Xeon 处理器，是使用系统总线消息传送机制和协议，通过系统总线在本地 APIC 和 I/O APIC 之间传送消息的。

P6 系列和 Pentium 处理器，是按照如下方式，通过串行 APIC 总线在本地 APIC 和 I/O APIC 之间传送消息的。因为 APIC 总线一个时间只能传送一个消息，所以，I/O APIC 和本地 APIC 就利用“轮换优先权”仲裁协议来获得在 APIC 总线上发送消息的权限。一个或多个 APIC 可能会并发地发送它们的消息。在每一个消息的开始，每个 APIC 都

会提出它正在发送消息的类型和它在 APIC 总线上的当前仲裁优先权。这个信息供仲裁之用。在每次的仲裁周期之后（在一次仲裁回合中），只有潜在的胜利者来驱动总线。到所有仲裁周期完成时，将会只剩下一个 APIC 在驱动总线。一旦选中了胜利者，就会赋予它排他性的使用总线的权利，和继续驱动总线发送它的实际消息的权利。

每次成功传输消息之后，所有 APIC 都会把它们的仲裁优先权提高 1。以前的胜利者（也就是那个成功传输消息的）的优先权假定为 0（最低）。仲裁期间那个优先权为 15（最高）但没有发送消息的代理，将采纳前一个胜利者的仲裁优先权，并增加 1。

注意，上面描述的仲裁协议会稍微有点不同，如果其中一个 APIC 发出一个特殊的中断结束（EOI）的话。这个高优先权的消息会获得总线，而不管它的发送者的仲裁优先权，除非有超过一个的 APIC 并发地发出 EOI 消息。在后一种情况，APIC 使用它们自己的仲裁优先权发送 EOI 消息。

如果 APIC 被设置来使用“最低优先权”仲裁（参看 8.6.2.4.“最低优先权传送模式”),并且多个 APIC 当前正在最低优先权上执行(APR 中的值),则使用仲裁优先权(Arb ID 寄存器中的唯一值）来打开这个结。APR 的所有 8 位都被用于最低优先权仲裁。

8.10.1.总线消息格式

有关用于串行 APIC 总线上传输消息的总线消息格式的描述，参看附件 F“APIC 总线消息格式”。

8.11.消息引发中断信号

《PCI 本地总线规范，版本 2.2》（www.pcisig.com）引进了消息引发中断信号的概念。当前具有这个能力的 Intel 处理器和芯片组是 Pentium 4 和 Intel Xeon 处理器。正如规范中所指出的：

“消息引发中断信号（MSI）是一个可选特征，它通过写一个系统定义的消息给一个系统指定的地址（PCI 的 DWORD 内存写事务），来开启 PCI 设备请求服务。在事务数据指定消息的同时，该事务地址指定消息目的。在设备配置期间，由系统软件去初始化消息目的和消息，分配一个或多个非共享的消息给每个具备 MSI 能力的功能。”

《PCI 本地总线规范》提供的能力机制用来识别和配置具备 MSI 能力的 PCI 设备。其它域中，此结构包含一个消息数据寄存器和一个消息地址寄存器。为请求服务，PCI

设备功能写消息数据寄存器的内容到消息地址寄存器中包含的地址处（和 64 位消息地址的消息上半截地址寄存器）。

8.11.1.和 8.11.2.提供了消息地址寄存器和消息数据寄存器的结构细节。该设备发出的操作是一个带有消息数据寄存器内容的对消息地址寄存器的写命令。该操作跟随语义规则之后，是为 PCI 写操作定义的，并且是一个 DWORD 操作。

8.11.1.消息地址寄存器格式

消息地址寄存器（低半截的 32 位）的格式如图 8-23 所示。

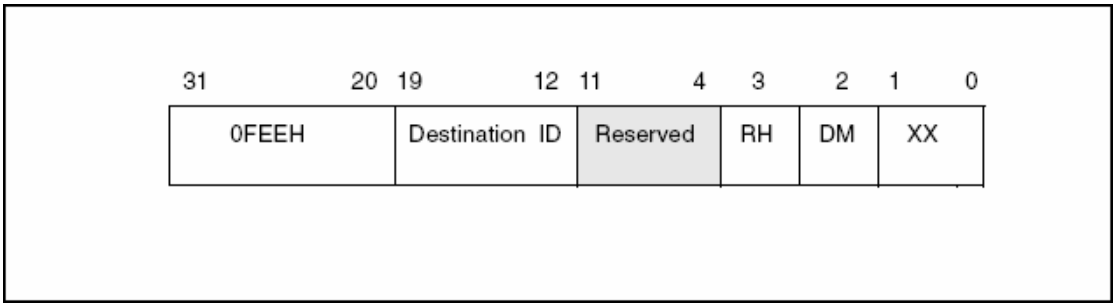

图 8-23 MSI 消息地址寄存器的结构

消息地址寄存器的域如下：

1．第 31 到 20 位：这些位包含一个中断消息的固定值（0FEEH）。这个值确定中断在基地址 4G-18M 开始一个 1MB 区域内。所有对这个区域的访问都被定向为中断消息。注意要确保没有其它设备把这个区域定义为 I/O 空间。

2．目的 ID：这个域包含 8 位目的 ID。它标识消息的目标处理器。如果 IOAPIC 用来调度中断给处理器的话，则该目的 ID 与 I/O APIC 重定向表项的第 63 到 56 位相对应。

3．重定向线索提示（RH）：这个位表明消息是否应该被定向到一个处理器那里，而这个处理器是可以接受消息的处理器组中具有最低中断优先权的那个处理器。

- 当 RH 是 0 时，中断定向到目的 ID 域所列的处理器上。
- 当 RH 是 1 并且采用了物理地址模式时，目的 ID 域不必设成 0xFF；它必须指向一个存在的并能够接收中断的处理器。
- 当 RH 是 1，并且逻辑目的模式活跃在一个使用平坦寻址模型的系统中时，则目的 ID 域必须被置位，以识别存在并且可以接收该中断的处理器组。
- 如果 RH 置为 1，并且逻辑目的模式活跃在一个使用集群寻址模型的系统中时，

则目的 ID 域不必设为 0xFF；用这个域标识的处理器必须存在并且能够接收中断。

4．目的模式（DM）：该位指出目的 ID 域是否应该被解释为最低优先权中断传送的逻辑或者物理 APIC ID。如果 RH 是 1 并且 DM 是 0，则目的 ID 域就在物理目的模式中，并且只有系统中有匹配 APIC ID 的处理器才被考虑传送那个中断（这意味着没有重定向）。如果 RH 是 1 并且 DM 是 1，则目的 ID 域就在逻辑目的模式中解释，并且重定向仅限于那些基于处理器的逻辑 APIC ID 和消息目的 ID 域的逻辑处理器组中的逻辑处理器。逻辑处理器组由那些与 8 位目的 ID 相匹配的逻辑目的构成，这些逻辑目的由每一个本地 APIC 的目的格式寄存器和逻辑目的寄存器标识。

8.11.2. 消息数据寄存器格式

消息数据寄存器的格式如图 8-24 所示。

保留的域没有假定任何值。软件在写的时候必须保留它们的内容。消息数据寄存器的其它域描述如下。

1．向量：该 8 位域包含与消息相关的中断向量。值的范围是 010H 到 0FEH。软件必须保证这个域不被编程为向量 00H 到 0FH。

2．传送模式：该 3 位的域指定中断接受如何处理。传送模式仅与指定的触发模式一同操作。正确的触发模式必须由软件保证。限制列示如下：

a．000B（固定模式）——传送信号给目的中列出的所有代理。固定传送模式的触发模式可以是边极的或者电平的。

b．001B（最低优先权）——传送信号给一个代理，它正在执行在目的域列出的所有代理中的最低优先权上。触发模式可以是边极的或者电平的。

c．010B（系统管理中断或者 SMI）——传送模式只是边极的。对于依赖于 SMI 语义的系统来说，向量域被忽略，但是，为了将来的兼容，必须编程设为全 0。

d．100B（NMI）——传送信号给目的域中列出的所有代理。向量信息被忽略。不管触发模式设置是什么，NMI 是一个边极触发的中断。

e．101B（INIT）——传送信号给目的域中列出的所有代理。向量信息被忽略。不管触发模式设置是什么，INIT 是一个边极触发的中断。

f．111B（ExtINT）——传送信号给目的域中列出的所有代理的 INTR 信号（就好

像源自一个 8259A 兼容的中断控制器的一个中断)。向量是通过激活 ExtINT 发出 INTA 周期来提供的。ExtINT 是一个边极触发的中断。

3．电平：边极触发中断消息通常被解释为激活消息。对于边极触发中断来说，这个域没有使用。对于电平触发中断来说，这个位反映出中断输入的状态。

4．触发模式：这个域指出将要触发一个消息的信号类型。

a．0——指出边极感应的。

b．1——指出电平感应的。

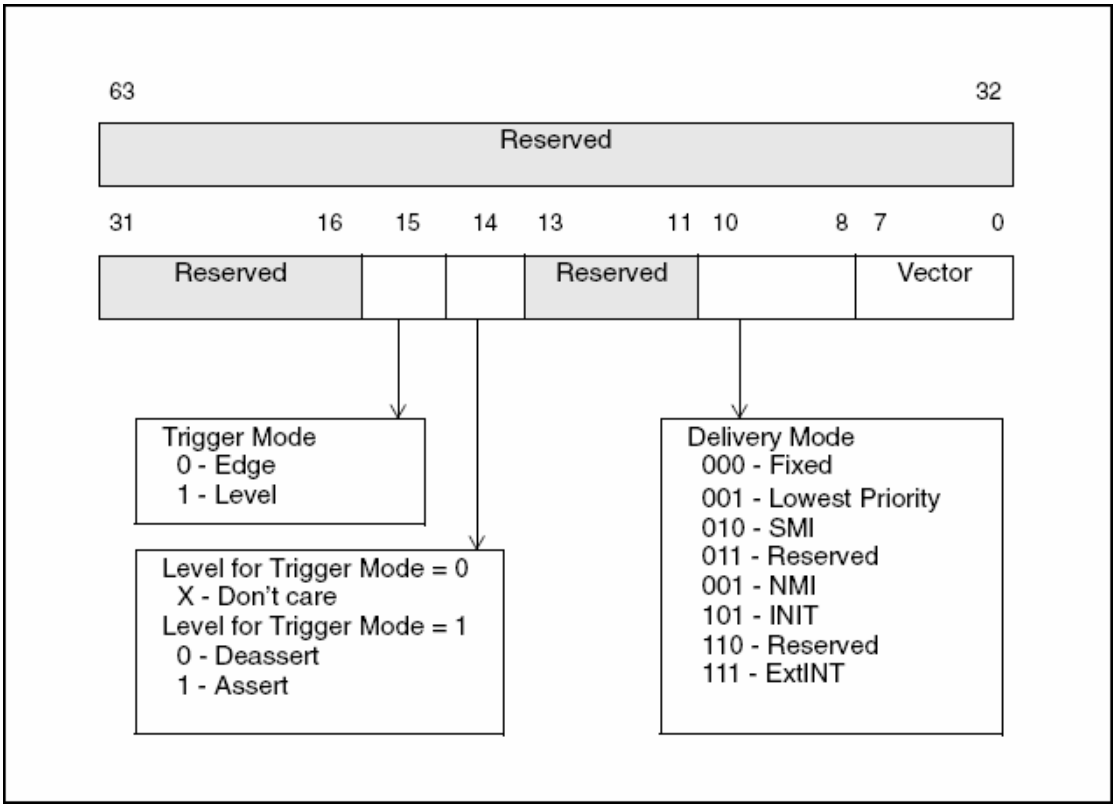

图 8-24 MSI 消息数据寄存器的格式

后　记

由 lijshu 发起倡议，借助于赵炯博士的 www.oldlinux.org 平台的交流和沟通，众多同道者共同努力，完成了这部《IA-32 架构第 3 卷：系统编程指南》部分章节的翻译。

担任各章翻译任务的网友如下：

第 1 章　导读	lijshu 翻译
第 2 章　系统架构概况	lijshu 翻译
第 3 章　保护模式内存管理	sportsman 翻译
第 4 章　保护机制	sportsman 翻译
第 5 章　中断和异常处理	wykr3879 和 blaze99 翻译
第 6 章　任务管理	wykr3879 和 blaze99 翻译
第 7 章　多处理器管理	Timeless 翻译
第 8 章　高级可编程中断控制器	blaze99 翻译

目前已经完成的部分由 blaze99 统稿。

由于我们水平的限制，译稿肯定有很多有待商榷的地方，错误也是不可避免的，欢迎大家批评指正。希望更加深入细致地了解 IA-32 架构的具体特点和细节的人，请查看原英文资料。

Blaze99

tongye2006@gmail.com

2005-12-10