IA-32 Intel®架构软件开发人员手册
卷 3：系统编程指南
(中文版-部分)

前言

现在开放源代码逐渐成为趋势的环境下，获取高水平的源代码的途径越来越容易，尤其是涉及操作系统的源代码，受到越来越多的有志于研究底层的开发人员的青睐。然而操作系统(内核)源代码具有规模大、专业性强、涉及的知识面广的特点，大部分首次接触内核源代码的人感觉不少部分尤其是与硬件平台相关部分(任务切换、内存管理等方面)难以理解，而这些难以理解的部分却又往往是操作系统的核心部分。

对于造成操作系统代码难以理解的原因有多个方面，主有对操作系统若干理论、概念理解不深，这可以通过阅读操作系统相关书籍来弥补。目前比较经典的操作系统书籍有多种，各有各的特色，有纯理论的，也有理论与实践相结合的。“工欲善其事，必先利其器”在开始探究操作系统源代码之前，仔细深入的研究这些基本概念、基本理论是十分必要的。然而这些还是不够的，除了理解操作系统概念理论之外，对于操作系统运行的硬件的了解也是非常必要的。而目前缺乏x86 平台权威资料，流行的教材都讲的比较基础，而与操作系统设计与开发方面相关的信息，讲得普遍比较少甚至少有涉及，而intel官方出的三卷手册就目前来讲是最全面、最权威的x86 平台资料了，由于是官方版本是英文版，所以在很大程度上限制了它的流行，即使平时查阅相关资料时，也大都只是参考它的部分章节而对其全貌仍未了解。这三卷各有特色，其中卷 3 主要是指针对与操作系统设计方面并鉴于目前情况，产生了首先将手册之卷 3 翻译为中文的念头，但是由于它篇幅很大(PDF版本有 780 页之巨)仅个人力量很难完成，所以借助于网络www.oldlinux.org平台，召集大家共同分担完成，在此也非常感谢赵博士为大家提供了那么好的交流平台。

目前已经分配的翻译任务如下：

第 1 章	关于本手册	
第 2 章	系统架构概况	我(lijshu)基本已经译完，就是目前这个文件:-)
第 3 章	保护模式下的内存管理	由 sportsman 负责翻译
第 4 章	保护机制	由 sportsman 负责翻译
第 5 章	中断和异常处理	由 wykr3879 负责翻译
第 6 章	任务管理	由 wykr3879 负责翻译
第 7 章	多处理器管理	由 Timeless 负责翻译
第 8 章	高级可编程中断控制器	由 beyond 负责翻译
第 9 章	处理器管理与初使化	由 极速时空 负责翻译

第 10 章 内存高速缓冲存储器控制　　由 engumen 负责翻译

第 11 章 Intel MMX 技术系统编程　　由 allen76312 负责翻译

第 12 章 SSE 和 SSE2 系统编程

第 13 章 系统管理　　由 victor011 负责翻译

第 18 章 IA32 处理器的兼容性　　由 marffin 负责翻译

其余的部分还没有落实，希望有兴趣、有精力的参加进来，让我们共同完成这个项目!

由于翻译是一件非常不容易的工作，尤其是达到“信、达、雅”的地步更就难了，译稿只能尽最大能力保持准确把握原文的意思，但是由于每个人能力及对原文的理解不同，因此对于译稿肯定有很多有待商榷的地方，甚至是错误的地方，因此请大家指出来，便于进一步修改、完善译稿，以供大家飨用。

lijshu
E-mail: lijshu@yahoo.com.cn
lijshu@hotmail.com

2005-1-13

第 1 章 关于本手册

IA-32 Intel®架构软件开发人员手册 卷3：系统编程指南(订单号245472)，它是三卷中其中的一部分，这三卷描述了IA-32 intel所有处理器的架构与开发环境。其它二卷是

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

卷1*基本架构*描述了IA-32的基本架构与开发环境,卷2*指令集参考*描述了处理器的指令集和操作码结构。这两卷是针对在操作系统下开发的应用开发人员，卷3*系统编程指南*描述了IA-32处理器对操作系统的支持，包括内存管理、保护、任务管理、中断和异常处理和系统管理模式。它也提供了关于IA-32处理器兼容的资料。这一卷是针对操作系统与BIOS的设计人员和开发人员的。

1.1 本手册包括的 IA-32 处理器种类

本手册主要适用于大多数最近的 IA-32 处理器，包括 Pentium® 、P6 系列处理器，Pentium4 处理器和 Intel® Xeon™处理器。P6 系列的处理器是指基于 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 IA-32 intel 架构概况 系统开发员指南，卷 3：系统开发指南

本手册包括以下内容：

第1章-关于本手册 介绍了IA-32 intel架构软件开发人员手册的三卷的内容，也描述了在这些手册中使用的符号约定以及与intel相关的手册和文档的列表,这些主要是针对于程序员和硬件设计人员。

第2章-系统架构概况 它描述了IA-32处理器的运行模式和IA-32架构对操作系统的支持，这些支持包括面向系统的寄存器和数据结构以及面向系统的指令。同时也讲述了从实模式到保护的切换所必需的步骤。

第3章-保护模式的内存管理 它描述了与分段及分页相关的数据结构、寄存器及指令，介绍了如何实现“平坦”(未分段)的内存模式或者分段的内存模式。

第4章-保护 它描述了IA-32架构中对分段保护所提供的支持。这一章也涉及了特权规则、栈切换、指针合法性检查、用户模式及管理模式。

第5章-中断和异常处理 它描述了IA-32架构定义的中断机制，介绍了与中断和异常相关的

保护以及架构是如何处理每一种异常类型的。在这一章末给出了每一种IA-32异常的相关资料。

第6章-任务切换 它描述了IA-32架构对多任务和任务间保护的支持。

第7章-多处理器管理 它描述了与多处理器相关的共享内存、内存调整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) 它描述了局部APIC的编程接口，给出了局部APIC与I/O APIC之间的接口。

第9章-处理器管理和初使化 它描述了IA-32处理器在复位(Reset)初使化之后的状态。这一章也描述了如何进入IA-32处理器的实模式和保护模式，以及如何在这二者之间进行切换。

第 10 章-内存高速缓存控制 它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。这一章也描述了内存类型范围寄存器(MTRRs- memory type range registers)，以及如何利用它们进行映射物理内存的内存类型。对于Pentium III、Pentium 4、和Intel Xeon处理器所引入的新的高速缓存控制和内存流指令这一章也有涉及。

第11章- Intel® MMX™技术系统编程 它描述了在进行与Intel MMX技术相关的系统编程时，所必须处理和考虑的几个方面，包括任务切换、异常处理和与现存系统环境的兼容等方面。Intel MMX技术是在IA-32架构中Pentium处理器引入的。

第12章-SSE和SSE2系统编程 它描述了SSE和SSE2扩展部分在进行系统编程时，所必须考虑的几个方面，包括任务切换、异常处理和与现存系统环境的兼容等。

第13章-系统管理 它描述了IA-32架构的系统管理模式(SMM- system management mode)和热量(thermal)监测方法。

第14章-机器检测(Machine-Check)架构 它描述了机器检查(machine-check)架构

第15章-调试和性能监测 它描述了IA-32架构中的调试寄存器和其它的调试机制。这一章也描述了时间戳计数器(time-stamp counter)和性能监测计数器

第16章-8080仿真 它描述了IA-32架构的实模式和虚拟8086模式

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

第18章-IA-32架构的兼容性 它描述了IA-32处理器之间的兼容性，包括intel286、intel386、intel486、Pentium、P6系列、Pentium 4、和Intel Xeon 处理器。P6系列包括Pentium Pro、PentiumII、and Pentium III 处理器。32位的IA-32处理器之间的差异，如

架构的一些专有特征，在这三卷中都有论述。这一章提供了与所有IA-32处理器兼容性相关的资料，描述了和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。Pentium处理器和P6系列的处理器事件也有描述。

附录B-模式相关寄存器(MSRs- Model Specific Registers) 列出了Pentium、P6系列、Pentium 4和Intel Xeon处理器中的MSRs，并描述了它们的功能。

附录C-P6系列处理器的MP初使化 给出了在MP系统中如何使用MP协议引导P6系列处理器的例子。

附录 D-LINT0 和 LINT1 输入编程 给出了如何使用 LINT0 和 LINT1 管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 给出了P6系列处理器的机器检查错误代码的解释。

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

1.3 IA-32 架构概况 软件开发人员手册，卷 1：基础架构

IA-32架构软件开发人员手册 卷1 的内容如下：

第1章-关于本手册 介绍了IA-32 intel 架构软件开发人员手册的三卷的内容，也描述了在这些手册中使用的符号约定与intel相关的手册和文档的列表,这些主要是针对于程序员和硬件设计人员。

第2章-IA-32 架构概况 本章介绍了IA-32 架构和基于本架构的处理器系列，同时也介绍了这些处理器一些共有的特征以及IA-32 架构发展的历史。

第3章-基本运行环境 介绍了内存管理的模式和应用程序使用的寄存器集合。

第4章-数据类型 描述了处理器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 指令集总汇 列出了所有IA-32 架构的指令，并根据指令所用的技术进行了分组(通用、x87FPU、intel MMX技术、SSE、SSE2和系统指令)，在这些组内指令按照各组的功能进行说明。

第6章-过程调用、中断和异常 描述了过程栈以及调用中断和异常服务的机制。

第7章-通用指令编程 描述了基本的装载、保存、程序控制、数学和字符串指令，这些指

令是基于基本数据类型和通用寄存器和段寄存器，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x87浮点单元(FPU)，包括浮点寄存器和数据类型，给出了浮点指令集简要介绍并描述了处理器的浮点异常产生的条件。

第9章-intel MMX技术编程 描述了Intel MMX技术包括MMX寄存器和数据类型，并给出了MMX指令集的概况。

第10章-SIMD扩展(SSE)编程 描述了SSE扩展，包括XMM寄存器、MXCSR寄存器和对齐(Packed)的单精度浮点数据类型，给出了SSE指令集的概况和访问SSE扩展的代码的书写方法。

第11章-SIMD扩展2(SSE2)编程 描述了SSE2扩展部分，包括XMM寄存器和对齐(packed)的双精度浮点数据类型，给出了SSE2指令集的概况和用指令访问SSE2扩展的方法，这一章也描述了SSE和SSE2指令产生的SIMD浮点异常，同时还给出了SSE和SSE2扩展部分对操作系统和应用代码的相互协作的方法

第12章-输入/输出 描述了处理器的I/O机制，包括I/O端口地址、I/O指令、I/O保护机制

第13章-处理器识别及其特征识别 描述如何识别CPU类型和和处理器中的特征。

附录A-EFLAGS 交叉引用 总结IA-32指令是如何影响EFLAGS寄存器的

附录B-EFLAGS条件码 总结如何根据条件代码标志中的(OF、CF、ZF、SF、和PF)标志进行条件跳转、传送和字节设置。

附录C-浮点异常总汇 总结了x87FPU浮点和SSE以及SSE2 SIMD浮点指令产生的异常。

附录D-编写x87FPU异常处理程序指南 描述如何设计和编写与MS-DOS兼容的FPU异常处理程序的方法，包括软件和硬件的所需条件以及汇编代码例子，这节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 介绍由SSE和SSE2 SIMD浮点指令所产生的异常处理程序的编写方法。

1.4 IA-32 架构概况 软件开发人员手册，卷 2：指令集参考

IA-32架构软件开发人员手册 卷2的内容如下：

第1章-关于本手册 介绍IA-32 intel 架构软件开发人员手册的三卷的内容，也描述了这些手册中使用的符号约定与intel相关的手册和文档的列表,这些主要是针对于程序员和硬

件设计人员。

第2章-指令格式 描述所有IA-32指令在机器级的格式，并给出了允许的前缀编码，操作数标识符字节(ModR/M字节)，和寻址方式字节(SIB字节)，以及转移和立即数字节

第3章-指令集参考 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。FPU和MMX指令都包括在这一章中。

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

1.5. 符号惯例

本手册对数据结构格式使用了特定的符号，对于指令的助记符，十六进制和二进制数字的表示也是如此，了解这些惯例就很容易阅读本手册。

1.5.1. 位和字节顺序

在内存数据结构的示意图中，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“小结尾”(little-endian)，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说明这种惯例：

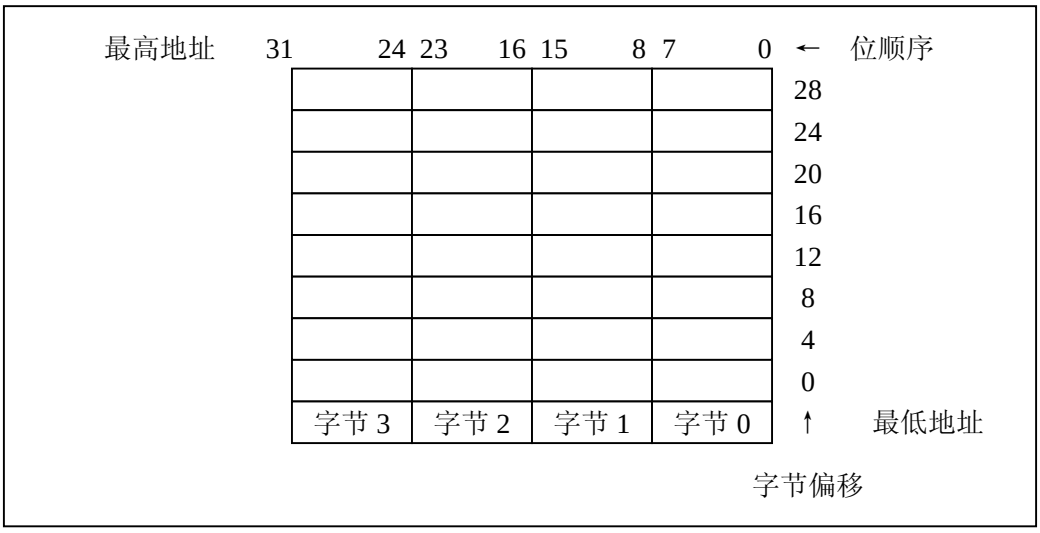

图 1-1 位与字节排列顺序

1.5.2. 保留位与软件兼容

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

- 在测试寄存器是否包含某些位时，不要依赖于任何保留位，在测试之前应该把这些位屏蔽掉
- 当把保存到内存或者保存到寄存器时，不要依赖于任何保留位

- 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

要避免软件依赖于IA-32中的保留位的状态,依赖于保留位将会导致软件就相当于依赖了一种不可预测的方式，这是由处理器处理这些位时的方式决定的。那些依赖于保留位的软件有可能与将来的处理器不兼容。

1.5.3 指令操作数

当用字符来代表指令时，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
- 参数 1、参数 2、参数 3 是可选的，根指令的不同可能有 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。参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寻址空间。

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寻址空间，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：下面的段地址代表DS寄存器指向的段中的FF79H地址

DS：FF79H

下面段地址代表代码段中的代码地址。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，该代码包含了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)

这个例子是一个缺页异常，这时的错误代码叫做错误类型。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，就像下面的通用保护异常

#GP(0)

对异常表示方法和相应的描述，请参看第5章，中断和异常处理。

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/processors/

这个站点列出的有些文档可以在线察看，有些可以在线订购。

文献资料是根据 intel 处理器和下面的类型列出的，即应用程序注意事项，数据表格、手册、论文和更新说明。

下面的文献可能会有用：

特定的 IA-32 处理器的数据表格

特定的 IA-32 处理器的更新说明

AP-485，Intel 处理器标识和 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Intel386处理器系列开始)为操作系统提供了广泛的支持。这些支持是IA-32系统级架构的一部分，包括下面的几个部分：

- 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

本章对IA-32系统架构提供了一个简要的介绍，在以后的几个章节中，分别对每个部分进行详细说明。这一章也描述了用来建立和控制系统的系统寄存器，并给出了处理器系统级(操作系统)指令的简要说明。IA-32系统级的架构特点只被系统开发人员使用，应用程序开发人员可能需要阅读这一章和下一章，在下一章中描述了架构特点的用法。这样便于理解系统开发人员对硬件的使用，以及通过这些使用为应用程序创建的安全可靠的环境。

注意：

这个概况和本书余下的部分主要集中介绍了IA-32的“原生”或者说是保护模式下操作。如第9章处理器管理和初使化一章所述，所有IA-32处理器在上电或者重启后首先进入实模式下，必须由软件进行由实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系统级操作的，比如内存管理、中断处理、任务管理和多处理器控制(多处理器技术)。图 2.1 给出了一个系统寄存器和数据结构的概况。

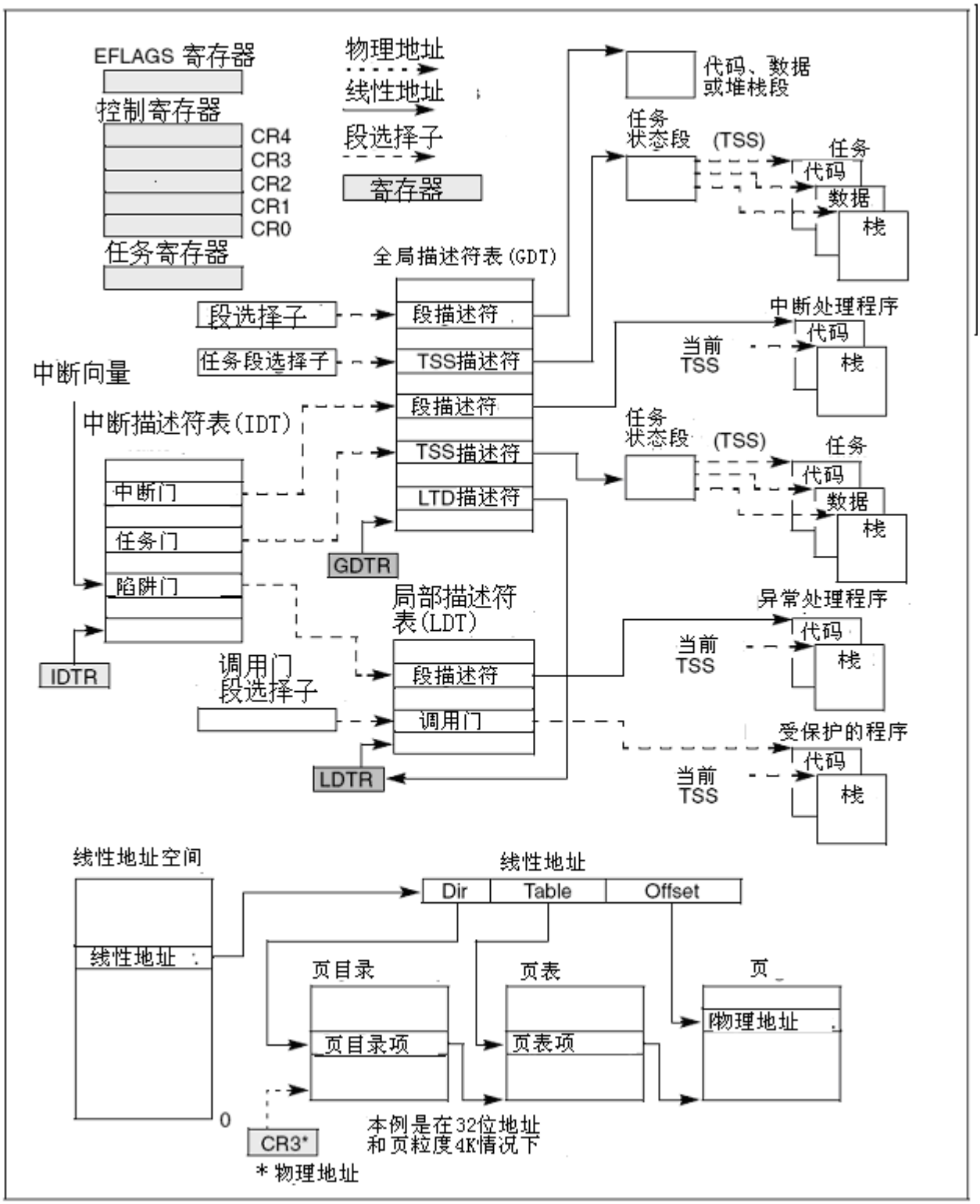

图2-1 IA-32系统级寄存器和数据结构

2.1.1.全局和局部描述符表

在保护模式下操作时，所有的内存访问要么通过全局描述符表(GDT)要么通过局部(可选)描述符表(LDT)，如图 2-1 所示。在这些描述符表里是段描述符，段描述符里包含了段的基地址、访问特权、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符。段选择符包含了 GDT 或 LDT(与它相关的段描述符)里的一个索引、一个全局/局部标志(决定段选择符是指向 GDT 还是指向 LDT)和访问特权等信息。要想访问段中的内容，必须同时提供段选择符和偏移地址，段选择符为段(在 GDT 或者 LDT 中)的描述符提供了一个访问途径。对于段描述符，处理器包含了线性地址空间里的基地址，偏移量确定了相对于基地址的字节地址。如果在处理器运行的当前特权级(CPL-Current Privilege Level)上可以访问段的话(CPL 被定义为当前执行代码段的保护级)，那么就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。

在图 2-1 中实心箭头代表线性地址，虚线箭头代表段选择符，点划线箭头代表物理地址。为了便于描述，许多段选择符被简化成直接指向段。然而实际上从段选择符到相应的段都是通过 GDT 或者 LDT。GDT 的线性地址是在 GDT 寄存器中(GDTR)，LDT 线性地址是在 LDT 寄存器中(LDTR)。

2.1.2. 系统段，段描述符和门

除了代码、数据和堆栈段是构成程序运行环境之外，系统架构还定义了两个系统段：任务状态段(TSS)和LDT。(GDT不被看作段因为它不能通过段选择符和段描述符访问)。这些段类型都有一个专门为它们定义的描述符。

系统架构也定义了一套称为门的描述符(调用门、中断门、陷阱门和任务门)，这些门提供了一种访问运行在不同于应用程序特权级的系统过程和处理程序的方法。例如一个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，调用程序必须提供调用门的选择符。执行访问特权检查的处理器比较调用门的特权和调用门指向的目的代码的 CPL。如果允许访问，处理器从调用门得到目标代码段的选择符和偏移地址。如果调用需要进行特权级的改变，处理器也切换到那个级别的堆栈(新堆栈的段选择符是通过当前运行任务的 TSS 获得的)。调用门也使得 16 位和 32 位之间的转换变得更加容易，反之亦然。

2.1.3. 任务状态段和任务门

TSS(如图 2-1)定义了任务执行环境的状态。这些状态包括通用寄存器、段寄存器、EFLAGS 寄存器、EIP 寄存器和段选择符以及三个堆栈段(特权 0、1、2 各一个堆栈)的指针的状态。它也包括了与任务相应的 LDT 的选择符和页表的基地址。

所有运行在保护模式下程序，都是一个称作当前任务的上下文中进行的。当前任务的 TSS 的段选择符保存在任务寄存器中。切换到一个任务的最简单的方法是进行 CALL 或 JMP 到那个任务中。新任务的 TSS 的段选择符是通过 CALL 或 JMP 指令给出。在进行任务切换时，处理器按照下面的次序进行：

1. 保存当前 TSS 中当前任务的状态
2. 装载新任务段选择符的任务寄存器
3. 通过 GDT 中段选择符访问新的 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。
5. 开始执行新任务

任务也可以通过任务门访，任务门与调用门很相似，除了它是提供(通过选择符)对 TSS 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。IDT包含了访问中断和异常处理程序的门描述表的集合。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中的门描述符包括有中断-、陷阱-、或任务门类型。在运行中断或异常处理程序时，处理器必须先从内部硬件、外部中断控制器、或通过执行INT，INTO，INT 3或BOUND指令的软件中断中接到一个中断向量(中断数字)。中断向量包含了IDT中的门描述符的索引。如果选中的门描述符是一个中断门或者陷阱门，相应的处理程序是通过非常类似于通过调用门调用过程了。如果描述符是一个任务门，处理程序是通过任务切换进行的。

2.1.5 内存管理

系统架构支持直接物理地址内存或者虚拟内存（通过分页）。当用直接物理地址时，线性地址就是物理地址，当使用分页时，所有代码、堆栈、系统段、GDT、IDT都可以将最近访问过页驻留在内存中而进行分页。页（在IA-32架构有时被称作页框）在物理内存中的位置保存在两个类型的系统数据结构中（页目录和页表），这两个数据结构都保存物理内存中（如图2－1）。页目录包含有页表的物理地址、访问特权、内存管理信息，页表中包含有页框的物理地址、访问特权和内存管理信息。页目录的基地址保存在控制寄存器CR3中。为使用分页机制，一个线性地址被分为三个部分：页目录、页表和页框中的偏移量。一个系统可以有一个或者多个页目录，比如每个任务都可以有自己的页目录。

2.1.6. 系统寄存器

为了有助于初使化处理器及控制系统的运行,架构在EFLAGES寄存器内提供了系统标志和几个系统寄存器：

- EFLAGES寄存器内的系统标志和IOPL域，控制着任务和模式切换、中断处理、指令跟踪和访问特权。2.3节的“系统标志和EFLAGES寄存器的域”对这些标志有详细的描述。
- 控制寄存器（CR0、CR2、CR4）包含了若干标志和数据域用于控制系统级的操作。这些寄存器内的其它标志指明了操作系统对处理器的兼容。2.5节“控制寄存器”有这些标志的描述。
- 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”有对这些寄存器的描述。
- GDTR、LDTR和IDTR寄存器内包含了各个表的线性地址和尺寸（界限）。2.4节“内存管理寄存器”有对这些寄存器的描述。
- 任务寄存器包含了当前任务的TSS的线性地址和界限。2.4节“内存管理寄存器”有对这些寄存器的描述。
- 模式相关的寄存器（图2-1内没有列出）

模式相关寄存器(MSRs)是一组主要用于操作系统的寄存器(也就是代码运行在0级特权下)。这些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（MTRRs）这些寄存器的个数和功能，在IA-32架构系列的处理器中的各处理器各有不同。9.4节“模

式相关寄存器(MSRs)”，MSRs更详细的信息在附录B“模式相关寄存器（MSRs）”中，那儿列出了完整的MSRs。

大多数的系统都限制应用程序访问所有的系统寄存器（而不是 EFLAGES 寄存器）。然而系统也可以设计成所有程序均运行在最高特权（0 级）上，在这种情况下应用程序可以修改所有系统寄存器。

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，系统架构还提供了下面的资源：

- 操作系统指令（参看2.6节“系统指令总汇”）
- 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，它可以编程用来记录诸如指令解码个数、接收的中断个数或者高速缓存装载次数等。15.8节“性能监测概况”对这些计数器进行了详细的讨论。

处理器提供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，缓冲区用来保存比如解码的系统地址和应用程序段以及等待的写操作等。第10章“内存高速缓存控制”详细讨论了处理器的高速缓存和缓冲区。

2.2.运行模式

保护模式 保护模式是处理器的原生模式，在该模式下，涵盖了处理器所有的特点和指令，有着最好的性能。对所有新应用程序和操作系统推荐使用该模式。

实模式 这个模式提供了intel8086的编程模式和一些扩展（比如切换到保护模式或系统管理模式）

系统管理模式（SMM）系统管理模式（SMM）在所有的IA-32体系中是一个标准的架构特征，它首先在intel 386SL处理器中出现。这种模式为操作系统实现电源和OEM专有特征提供的一种透明的机制。SMM模式是通过激活外部系统中断针（SMI＃）而进入的，激活产生了一个系统中断（SMI）。在SMM中处理器先保存好当前运行的程序和任务的上下文，然后切换到一个单独的地址空间，从SMM返回后处理器再返回SMI之前的状态。

虚拟 8086 模式 在保护模式中，处理器提供了一种准模式叫作虚拟 8086 模式。这种模式允许在多任务的保护模式下处理执行 8086 程序。

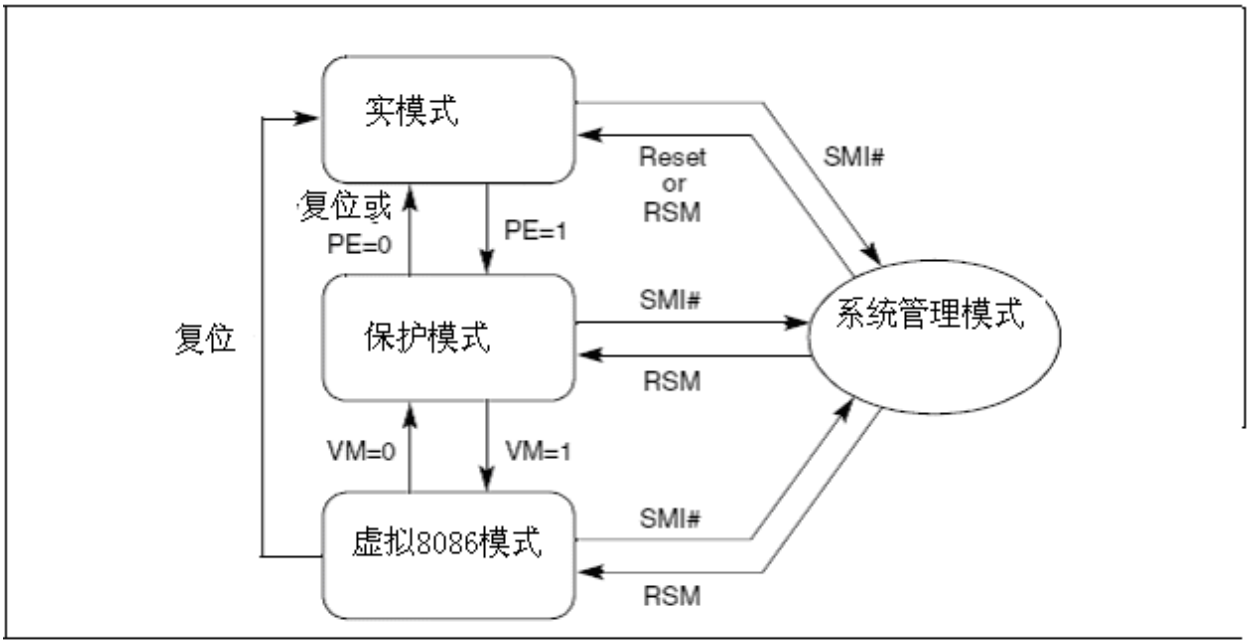

图2-2 处理器运行模式之间的转换

处理器在上电或重启后自动进入到实模式，控制寄存器 CR0 中的 PE 标志控制着处理是在实模式下还是在保护模式下（2.5 节“控制寄存器”）。9.9 节“模式切换”详细介绍了实模式和保护模式之间的切换。EFLAGES 寄存器中的 VM 标志决定了处理器是在保护模式下还是在虚拟 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.节“进入虚拟 8086 模式”）。

不论处理器是处在实模式还是处在保护模式、虚拟 8086 模式下，只要接收到 SMI 它就切换到 SMM 模式，执行完 RSM 指令，处理器再返回进入 SMI 模式之前的模式。

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（通常是操作系统代码）可以修改这些位，系统标志和IOPL的作用如下：

TF　　**陷阱（第8位）**置1是调试状态下的单步执行，置0是禁用单步执行。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指令修改TF标志，那么调试异常就在执行POPF、POPFD或者IRET指令后产生。

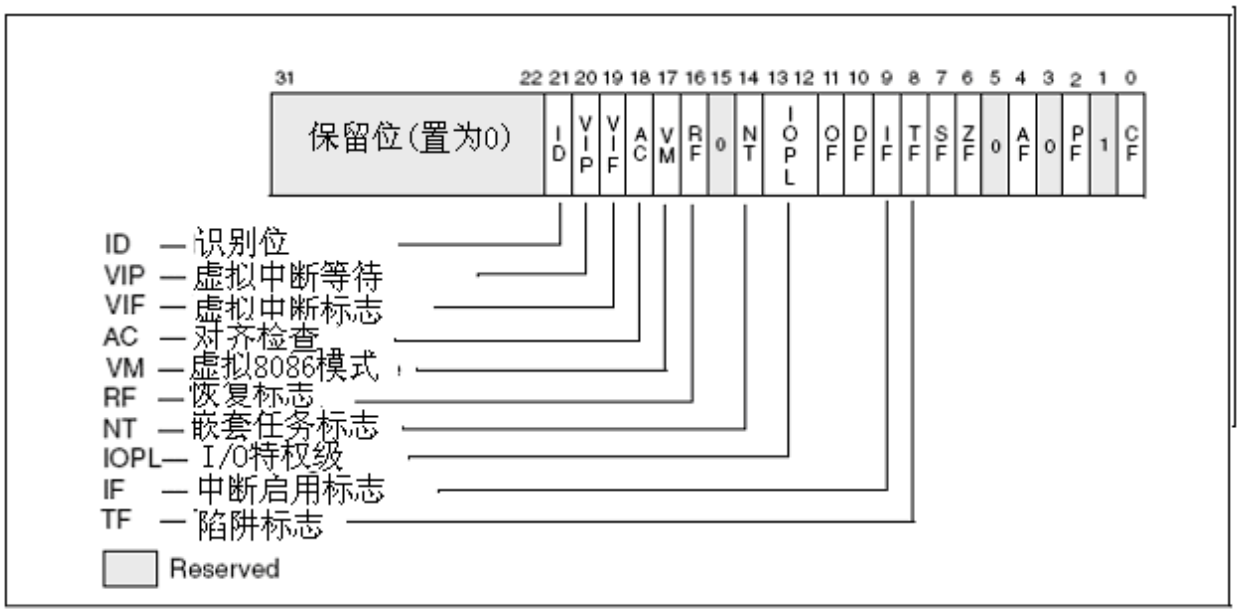

图2-3 EFLAGES寄存器中的系统标志

IF **中断允许(位9)**控制着处理器对可屏蔽硬件中断(见5.3.2.节“可屏蔽硬件中断”)请求的响应。置1是响应可屏蔽硬件中断，置0为禁止响应可屏蔽硬件中断，IF标志并不影响异常和不可屏蔽中断（NMI）的产生。控制寄存器CR4中的CPL、IOPL和VME标志决定着IF标志是可否可以由指令CLI、STTI、POPF、POPFD和IRET修改。

IOPL **I/O特权域（位12和位13）**指出当前程序或任务的I/O特权级别。当前程序或任务的CPL必须小于或等于IOPL才可以访问I/O地址空间。当运行在0级特权时，该域只能由的POPF和IRET指令修改。参见第12章 *输入/输出 IA-32intel架构软件开发人员手册，卷1*里有对IOPL和I/O操作之间的关系详细的介绍。当虚拟模式扩展起作用时(控制寄存器CR4中的VME置位时),IOPL也是控制IF标志的修改以及控制虚拟8086模式下中断的处理方式的机制之一。

NT **嵌套任务（位14）**控制被中断和被调用的任务的链接。处理器在调用一个由CALL指令、中断或者异常触发的任务时设置该位。当任务因调用IRET指令而返回时，处理器检测并修改该位。该标志可以由POPF/POPFD指令直接置位或清零，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节“任务链”对嵌套任务的详细描述。

RF **恢复（位16）**控制着处理器对断点指令条件的响应。当置1时，该标志可以临时禁用由于指令断点而产生调试异常（＃DE），但是其它的异常条件仍可以产生异常。置0时指令断点产生调试异常。

RF标志的主要功能是重新执行由指令断点而引发的调试异常后面的指令。调试器软件必须在程序调用IRET指令返回之前，将栈中的EFLAGES映象该位置为1，以阻止指令断点产生另外的调试异常。在返回到已成功执行的指令之后，处理器会自动地将该位清零，从而可以继续产生指令断点。

参见15.3.1.1.“指令断点异常条件”有对该标志用法的详细介绍。

VM　**虚拟8086模式（位17）**置1进入虚拟8086模式，置0返回保护模式。参见16.2.1.节“启用虚拟8086模式”里对该标志切换到虚拟8086模式的详细介绍。

AC　**对齐检查（位18）**。将该位置1的同时，将控制寄存器中CR0中的AM标志置1就启用了对内存引用的对齐检查。将AC标志和/或AM标志清零就禁用了对齐检查。当引用一个没有对齐的操作数时，将会产生一个对齐检查的异常，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模式（3级特权）下产生。默认特权为0的内存引用，比如段描述表的装载，并不产生这个异常，虽然它在用户模式会产生。对齐检查异常可以用于检查数据的对齐，这对于当和其它处理器交换数据时是有用的，交换数据需要所有数据对齐。对齐检查异常也可以被解释程序用来将某些指针标记不对齐从而成为特殊指针，这样就减轻了对每个指针进行对齐检查的负担，只要对使用的特殊指针进行就可以了。

VIF　**虚拟中断（位 19）**包含了一个 IF 标志的虚拟映象。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）。

参见16.3.3.5.节“方法6：软件中断处理”和16.4节“保护模式虚拟中断”关于本标志的详细信息。

VIP　**虚拟中断等待(pending)（位20）**由软件置1表明有一个中断是正在等待被处理，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）。

参见16.3.3.5“方法6：软件中断处理”和16.4节“保护模式虚拟中断”关于本标志的详细信息。

ID　**识别（位21）**软件置1或0表明是否支持CPUID指令

2.4.内存管理寄存器

处理器提供了4个内存管理寄存器（GDTR、LDTR、IDTR和TR），这些寄存器指明了那些控制分段内存的数据结构的位置（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寄存器保存了GDT的32位基地址和16位表界限。基地址是指GDT的0字节的线性地址，表界限是指表中的字节个数。LGDT和SGDT指令是用来分别装载和保存GDTR寄存器的。处理器一上电或复位，基地址就被设为缺省的0，表界限设为FFFFH。对于保护模式的操作，作为处理器初使化过程的一部分，一个新的基地址必须装入GDTR。参看3.5.1节“段描述符表”关于基地址和界限域的详细说明。

2.4.2 局部描述符表寄存器（LDTR）

LDTR寄存器保存了16位段选择符、32位基地址、16位段界限和LDT描述符属性。基地址是指LDT段的0字节的线性地址，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指令是专门分别用来装载和保存LDTR寄存器段选择符那部分的。包含LDT的段必须在GDT中有一个段描述符。当LLDT指令装载一个LDTR中的段选择符时，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，LDTR就会自动被装载连同新任务的段选择符和描述符。在写新的LDT信息到寄存器前，LDTR的内容前并不会自动的保存。处理器一上电或复位，段选择符和基地址都被设缺省的0，

界限被设为FFFFH

2.4.3 IDTR 中断描述符表寄存器

IDTR寄存器保存了IDT的32位基地址和16位表界限。基地址是指IDT的字节0的线性地址，表界限是指表中的字节个数。LIDT和SIDT是专门分别用来装载和保存IDTR寄存器的指令。处理器一上电或复位，基地址就被设为缺省的0，界限就被设为FFFFH。作为处理器初使化过程的一部分，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（IDTR）”关于基地址和限域的详细说明。

2.4.4.任务寄存器（TR）

任务寄存器保存着16位的段选择符,32位基地址,16位段界限和当前任务的TSS描述符属性。它引用GDT中的TSS描述符。基地址指明TSS中的0字节的线性地址，段界限指明TSS中的字节个数。（参见6.2.3.节“任务寄存器”有关于任务寄存器的详细描述）

LTR 和 STR 指令是分别用来装载和保存任务寄存器段选择符部分的。当用 LTR 装载一个任务寄存器中的段选择符时，基地址、界限和 TSS 描述符都被自动的装载到任务寄存器。处理器上电或复位后，基地址设成默认的 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符和 TSS 描述符。在往任务寄存器写新的内容时，任务寄存器并不会自动保存。

2.5 控制寄存器

控制寄存器（CR0、CR1、CR2、CR3和CR4见图2-5）决定了处理器的运行模式和当前正在执行的任务的特征，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－包含缺页的线性地址（引起缺页的线性地址）

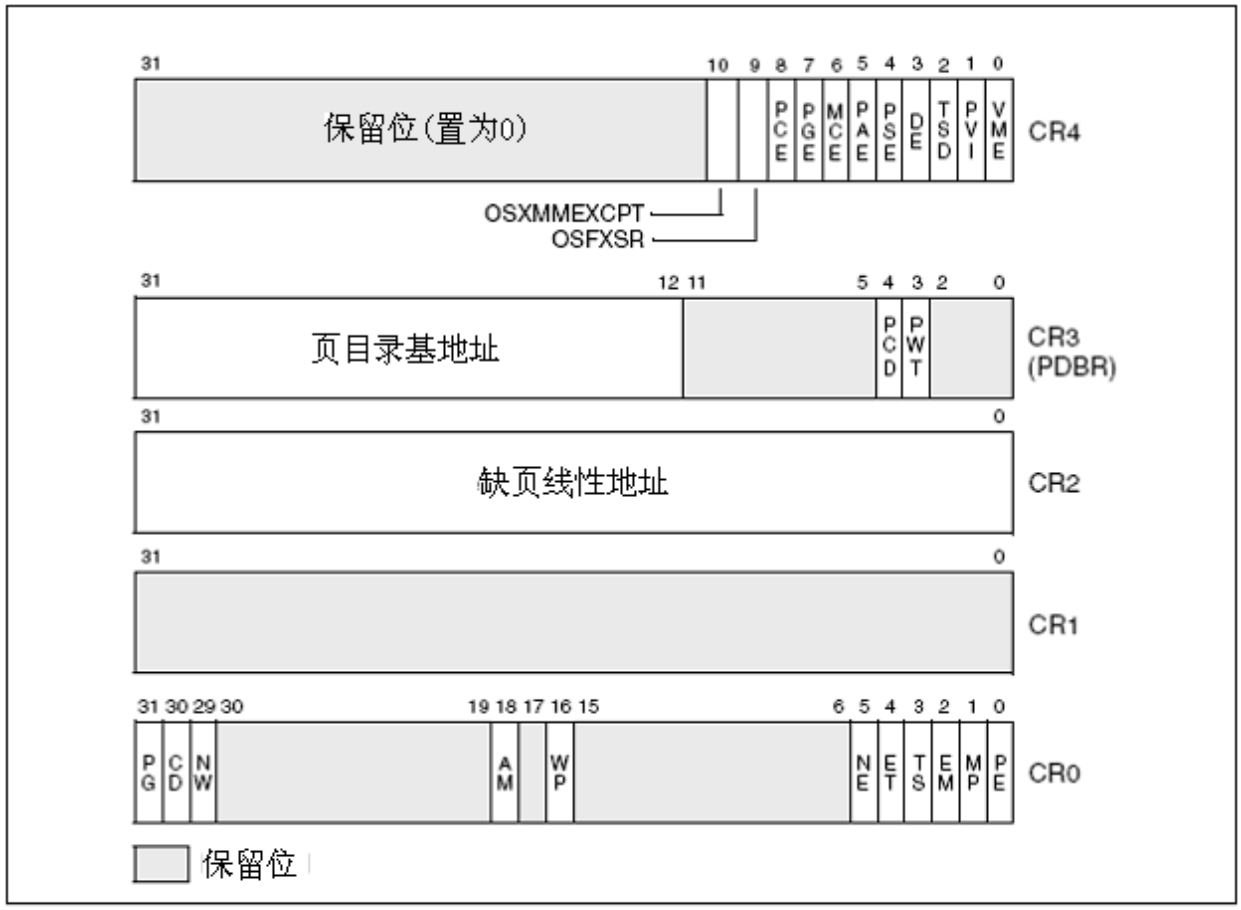

图2-5. 控制寄存器

CR3－包含了页目录的基地址和二个标志（PCD和PWT）。该寄存器也被称为页目录基地址寄存器（PDBR）。页目录基地址只有高20位确定，低12位是0，所以页目录地址必须是页边界对齐的（4K字节）。PCD和PWT标志控制着页目录在处理器内部数据缓冲区的缓存（它们不控制TLB页目录信息的缓存）。当使用物理地址扩展时，CR3寄存器包含了页目录指针表的基地址（见3.8节“使用PAE分页机制实现36位物理地址”）。

CR4－包含了一组标志，这些标志启用了架构方面的几个扩展，并指明了系统对某些处理器支持的能力。这个控制寄存器可以通过用MOV指令“从寄存器读或者写到寄存器”的方式进行读取或者装载（修改）。在保护模式下，MOV指令允许读取或者装载控制寄存器（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。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，所有的线性地址都可以当作物理地址对待。如果PE标志（CR0中位0）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节“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的位30）**当CD和NW标志为0时，处理器中内部（和外部）内存位置的高速缓存将启用。当CD标志置1时，高速缓存将会被禁用，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并且使缓存失效，这样就不会命中高速缓存了（见10.5.3节“阻止高速缓存”）见10.5节“高速缓存控制”对于选中的页或者内存位置的高速缓存限制的详细描述

NW　**不直写（CR0位29）**当NW和CD标志都置0时，回写（即write-back，主要是对 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）被用来写命中缓存时的数据，并且启用失效循环。表10-5详细介绍了NW标志在高速缓存方面对CD和NW设置的影响。

AM　**对齐屏蔽（CR0的位18）**置1时启用自动对齐检查，置0时禁用对齐检查。对齐检查只有在AM标志、EFLAGES中的AC标志置1时并且CPL是3、处理器运行在保护模式下或者虚拟8086方式下才进行，

WP　**写保护（CR0的位16）**置1时禁止管理级的过程往用户级只读页中写，置0时允许管理级的过程往用户级只读页中写。这个标志是用来在创建（forking）一个新进程时实现写拷贝（COW-copy on write），在UNIX操作系统中就是如此。

NE　**数值错误（CR0中的位5）**置1时启用原生的（内部的）x87FPU错误报告机制，置0时启用类PC的x87FPU错误报告机制。当NE标志置0且检查(ASSERT)IGNNE#输入时，一个未屏蔽处理的x87FPU错误，会引起处理器检查FERR#针来产生一个外部中断，并且在执行下一条等待浮点指令或WAIT/FWAIT指令之前，立即停止指令的执行。FERR#针是用来驱动输入到外部中断控制器的（FERR＃模拟intel 287和intel387DX数学处理器的ERROR#针的）。NE标志、IGNNE＃针和FERR＃针和外部逻辑一起被用来实现类PC的错误报告机制的（参见第8章“软件异常处理”和卷1中的附录D中关于x87 FPU错误报告机制，以及当FERR＃针在被检查(ASSERT)时，具体实现所依赖的东西）。

ET　**扩展类型（CR0的位4）**在Pentium 4、Intel Xeon、P6系列、和Pentium 处理器中为保留位，被硬编码为1。在intel 386和intel486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的位3）**允许当一个任务切换延迟至x87 FPU、MMX、SSE或SSE指令被新任务执行时，保存x87 FPU，MMX，SSE和SSE2的上下文。处理器在每次任务切换时设置该位，并且当执行x87 FPU，MMX，SSE和SSE2指令时测试该位。

- 如果TS标志置1并且EM标志（CR0的位2）置0时，那么在x87 FPU、MMX、SSE和SSE2指令执行前， “设备不可使用”异常（＃NM）会产生，但是这些指令并不包括PAUSE、PREFETCH*h*、SFENCE、LFENCE、MFENCE、MOVNTI和CLFLUSH指令。（参见WAIT/FWAIT指令下的相关叙述）
- 如果TS标志置1并且MP标志（CR0的位1）和EM标志都置0时，那么在执行x87 FPU WAIT/FWAIT指令之前＃NM异常并不会产生。
- 如果EM标志置1，TS标志的值对x87 FPU、MMX、SSE和SSE2指令的执行就没有什么影响。

表2-1列出了处理器遇到x87 FPU指令时，根据TS、EM、MP标志的值所作出的不同反应。表11-1和12-1列出了处理器分别遇到MMX和/或SSE或SSE2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。相反地处理器将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或SSE2 指令（前面列出的指令除外），就会引起＃NM异常。

＃NM异常处理程序可以用来清除TS标志（用CLTS指令）并且保存x87 FPU、XMM的上下文和MXCSR寄存器。如果任务从未遇到x87 FPU、MMX、SSE或者SSE2指令，那么x87 FPU、MMX、SSE和SSE2的上下文就从不保存。

表 2-1 **根据 EM、 MP and 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的位2）**置1时表明处理器没有内部或者外部的x87 FPU，置0时表明有x87 FPU。这个标志也影响MMX、SSE和SSE2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”的异常(#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根据IA-32处理器和x87 FPU或者系统中有的数学协处理器，列出了该标志的推荐值。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指令将会产生个非法操作码的异常(#ND)(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以便于MMX指令的执行。对于SSE和SSE2扩展也是一样，当EM为1时，大多数SSE和SSE2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和SSE2扩展，EM就必须置为0以便于运行这些指令。不受EM的值影响的SSE和SSE2指令有PAUSE、REFETCH*h*、SFENCE、LFENCE、MFENCE、MOVNTI和CLFLUSH指令。

MP　**监测协处理器(CR0的位1)**控制WAIT(或者FWAIT)指令与TS标志(CR0的位3)的相互作用。如果MP标志是1，WAIT指令将会“设备不可使用”的异常(#NM)如果TS标志是1。如果MP标志是0，WAIT指令就会忽略TS标志的值。表9-2列出了这个标志的推荐设置，这些设置是根据IA－32处理器和系统中是否有x87 FPU或协处理器而进行的。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的位0）**置1时启用保护模式，置0时启用实模式。这个标志并不直接启用分页机制。它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节“模式切换”中利用PE标志在实模式与保护模式之间进行切换。

PCD　**禁用页级缓存（CR3的位4）**控制当前页目录是否缓存。置1时禁止页目录缓存，置0启用页目录缓存。这个标志只影响处理器内部缓存（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（禁用缓存）标志置0，处理器将忽略这个标志。第10章内存缓存控制　中有关于这个标志的详细说明。参见3.7.6节“页目录和页表项”对PCD标志在页目录和页表项协作的详细信息。

PWT　**页级透明写（CR3中的位3）**控制着当页目录的直写或回写的缓存机制。如果PWT标志置1，则用直写，置0启用回写缓存。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（禁用

缓存）标志为1，处理器将忽略这个标志。参见10.5节“缓存控制”中对这个标志的详细介绍。参见3.7.6节“页目录页表项” 对PCD标志在页目录和页表项协作的详细信息。

VME **虚拟8086模式扩展（CR4中的位0）**置1时则在虚拟8086模式下，启用中断和异常处理扩展。置0时禁用扩展功能。虚拟模式扩展的应用是通过减少虚拟8086监控程序对8086程序执行过程中出现的中断和异常的处理，并且重定向中断和异常到8086程序的处理程序，从而改进虚拟8086模式下应用程序的性能。对于虚拟中断标志（VIF）它也提供了硬件支持来改进在多任务及多处理器环境下执行8086程序的可靠性。参见16.3“虚拟8086模式下的中断和异常处理”中对这一特征使用方法的详细论述。

PVI **保护模式下的虚拟中断（CR4中的位1）**置1时对于虚拟中断标志（VIF）在保护模式下启用硬件支持，置0时在保护模式下禁用VIF标志。参见16.4“保护模式下虚拟中断“中对这一特征用法的详细论述。

TSD **禁用时间戳（CR4中的位2）。**置1时将限制运行在0级特权下的程序执行RDTSC指令。置0时则允许任何特权程序执行这一指令。

DE **调试扩展功能（CR4的位3）**置1时，对调试寄存器DR4和DR5的引用会引起一个未定义的操作码(#UD)异常；置0时为了与运行在早期IA-32处理器上面的程序兼容，处理器会混淆对DR4和DR5的引用。参见15.2.2节“调试寄存器DR4和DR5”中对这一标志的详细说明。

PSE **页尺寸扩展(CR4中的位4)**置1时页大小为4M字节，置0时页大小为4K字节。参见3.6.1节“页选项”中对这个标志用法的介绍。

PAE **物理地址扩展(CR4中的位5)**置1时启用分页机制来引用36位物理地址；置0时只可引用32位地址。参见3.8节“用PAE分页机制实现36位物理地址访问”中对物理地址扩展的详细说明。

MCE **启用机器检测(CR4中的位6)**置1时启用机器检测(machine-check)异常，置0时禁用机器检测异常。参见第14章“机器检查架构”中对机器检查异常和机器检查架构的详细说明。

PGE **启用全局页(CR4中的位7)**(在P6系列处理器中引入)置1时启用全局页，置0时禁用全局页。全局页这一特征能够使那些经常被使用或共享的页对所有的用户标志为全

局的(通过页目录或者页表项中的第8位-全局标志来实现)。在任务切换或者往CR3寄存器写时，全局页并不从TLB中刷新。当启用全局页这一特征时，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)。如果将这个顺序颠倒了，可能会影响程序的正确性以及处理器的性能会受损。参见3.11节“转换查找缓冲区TLB”中对这一位的详细使用。

PCE　**启用性能监测计数器(CR4中的位8)**置1时，允许RDPMC指令执行，不论程序运行哪个特权级别。置0时RDPMC指令只能运行在0级特权上。

OSFXSR **操作系统对FXSAVE和FXRSTOR指令的支持(CR4中的位9)**置1时，这一标志具有下列功能：(1)表明操作系统支持FXSAVE和FXRSTOR指令(2)启用FXSAVE和FXRSTOR指令来保存和恢复XMM和MXCSR寄存器连同x87 FPU和MMX寄存器的内容(3)允许处理器执行除了PAUSE、PREFETCHh、SFENCE、LFENCE，MFENCE、MOVNTI和CLFLUSH指令之外的任何SSE和SSE2指令。如果这一标志置0，则FXSAVE和FXRSTOR指令保存和恢复x87 FPU和MMX寄存器的内容，但可能不保存和恢复XMM和MXCSR寄存器的内容。另外，如果这一标志置0，当处理器企图执行除了PAUSE、PREFETCHh、SFENCE、LFENCE、MFENCE、MOVNTI和CLFLUSH指令之外的任何SSE和SSE2指令时，都将会产生一个非法操作码异常(#UD)，操作系统必须正确地设置这一标志。

注意：

CPUID特征标志FXSR、SSE和SSE2(位24、25、26)分别表示在特定的IA-32处理器上，是否具有FXSAVE/FXRESTOR指令，SSE扩展以及SSE2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的SIMD浮点异常(CR4中的位10)，表明操作系统通过异常处理程序支持非屏蔽的SIMD浮点异常的处理，该异常处理程序在SIMD浮点异常产生时被调用。操作系统必须正确的设置这一标志，如果这一标志没有设置，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 都是与模式相关的。所有的这些标志(除了 PCE 标志)在使用之前都可以通过 CPUID 指令来检查它们处

理器是否已经实现。

2.6.系统指令总汇

系统指令是用来处理系统级的功能，比如装载系统寄存器、管理高速缓冲存储器、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执行(即运行在0级特权上)。其它的指令则是可以在任何特权级别上运行，应用程序也可以执行它们。表2-2列出了系统指令，并表明了它们对于应用程序是否有用以及能否执行。这些指令在卷2中的第3章指令集参考中有详细说明。

指令	指令描述	对应用程序是否有用	应用程序能否执行
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	
RSM	从系统管理模式返回	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．这个指令是在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中装载到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) IDTR寄存器的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段选择符和段描述符到LDTR。(段选择符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段选择符保存内存中或者通用寄存器中。

LTR(装载任务寄存器) 把TSS的段选择符和段描述符从内存中装载到任务寄存器中(段选择符操作数也可以位于通用寄存器中)。

STR(保存任务寄存器) 把当前任务的任务寄存器中的段选择符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。这些指令是为兼容16位intel 286处理器而提供的。运行在32位IA-32处理器上的程序不应该再使用这些指令，相反应该用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。这一指令允许TS标志在x87 FPU上下文保存以后清零，从而避免进一步出现#NM异常。参见2.5节“控制寄存器”中对这一标志的详细说明。

控制寄存器(CR0、CR1、CR2、CR3和CR4)都是用MOV指令来装载的。这一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用来检查段选择符和段描述符,看是否允许访问与它们相关联的段。这些指令自动进行访问特权和类型检查，与处理器的作法一样，这样就使操作系统阻止了异常的产生。ARPL(调整RPL)指令调整段选择符的RPL(请求访问特权)来匹配那些提供该段选择符的程序。参见4.10.4节“检查调用者访问特权(ARPL指令)”中对这些指令的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以及从段描述符中装载访问特权到通用寄存器中。软件可以检查访问特权，看看段类型是否与它将要用的兼容。参见4.10.1“检查访问特权(LAR指令)”中对这一指令的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。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节“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(读校验)和VERW(写校验)指令是分别校验选定的段，在某个CPL上是否可读的或可写的。参见4.10.2节“检查读/写特权(VERR和VERW指令)”中对这条指令的功能和用法的说明。

2.6.3 装载和保存调试寄存器

处理器的内部调试方法是由一组8位调试寄存器控制的(DR0到DR7)。通过MOV指令可以从这些寄存器中装载或保存。

2.6.4 使 Cache 和 TLB 无效

处理器有几条指令用于使Cache和TLB无效。INVD(使Cache无效，无回写)指令使内部Cache中的所有数据和指令无效，并给外部的Cache发送信号以指明它们也应该无效。WBINVD(使Cache无效，有回写)指令执行与INVD同样的功能，除了在使Cache无效之前它将内部Cache中修改的行回写到内存。使内部有Cache无效后，它给外部Cache发信号，让它们回写修改的数据并使它们的内容无效。INVLPG(使TLB无效)指令针对特定的页使TLB无效。

2.6.5 控制处理器

HLT(暂停处理器)指令暂停处理器直至接收到一个启用中断(比如NMI或SMI，这些都是启用中断)、调试异常、BINIT#信号、NINT#信号或RESET#信号。处理器产生一个特殊的总线循环以指明进入暂停模式。硬件对这个信号的响应有几种方式，前面板上的指示灯可能会打开，产生一个用于记录诊断的NMI中断。复位初使化被调用(注意BINIT#针是在Pentium Pro处理器引入的)。they will be handled after the wake event from shutdown is processed(比如A20M中断)。在修改内存操作时，LOCK前缀调用锁住的读-修改-写操作(原子的)。这个机制在多处理器系统中用于处理器之间进行可靠的通讯。在Pentium和早期的IA-32处理器中，LOCK前缀会使处理器在执行那些总是引起显式总线锁出现的指令时，检测LOCK#信号。在Pentium 4、Intel Xeon和P6系列处理器中，锁操作是通过一个Cache锁或总线锁来处理。如果内存访问是可以缓存的话，并且只影响一个单独的缓存线，那么就会调用缓存锁，系统总线和系统中内存中真正的内存位置在操作中不会被锁定。这里，其它的总线上的Pentium 4、Intel Xeon或者P6系列处理器回写所有的已修改数据并使它们的缓存无效，以保证系统内存的一致性。如果内存访问不能缓存且/或它跨越了缓存线的边界，那么这个处理器的LOCK#信号就会被检查并且处理器在上锁期间不会响应总线控制请求。RSM(从SMM返回)指令还原处理器(从上下文中)到系统管理模式(SMM)中断之前的状态。(这一段的翻译感觉不太准确，还有待商榷)

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。这些计数器可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式相关的，并且包含解码的指令条数，接收的中断个数、装载高速缓存的次数。每个计数器都能用来监测一个不同的事件，用系统指令WRMSR可以在45ESCR和18 CCCRMSRs ，或者在PerfEvtSel0 or the PerfEvtSel1 MSR(对P6系列处理器)写入数值。RDPMC指令从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式相关的64位计数器，每次处理器复位后，它都置为0。如果没有复位，处理器在200MHZ时钟频率下运行，计数器将每年增加~6.3×10^{15}。在这一频率下它将运行2000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节“性能监测概况”和15.7节“时间戳计数”中对性能监测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 读写模式相关寄存器

RDMSR(读模式寄存器)和 WRMSR(写模式相关寄存器)允许分别对处理器的 64 位模式相关寄存器(MSRs)进行读写。进行读写模式相关寄存器时，其值是在 ECX 寄存器中。RDMSR 指令把特定的 MSR 的值读取到 EDX：ECX 寄存器中；WRMSR 把 EDX：EAX 寄存器中的值写入特定的 MSR 中。参见 9.4 节“模式相关寄存器(MSRs)”中对 MSRs 的详细介绍。RDMSR 和 WRMSR 指令是在 IA-32 架构中的 Pentium 中引入的。

第 3 章 保护模式下的内存管理

（这一部分是由 sportsman 负责翻译的，感谢他的辛苦工作！）

本章描述了 intel 体系结构的保护模式内存管理，包括物理内存需求，分段机制和分页机制。关于处理器的保护机制的描述，请参考本书第四章。关于实模式下的内存地址和虚拟 8086 模式的保护的描述，请参考本书第 16 章。

3.1 内存管理概述

Intel 体系结构的内存管理可分为两部分：分段和分页。分段提供了一种机制，这种机制可以为每个程序或者任务提供单独的代码、数据和栈模块,这就保证了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的请求调页虚拟内存系统，在这种系统中，程序的执行代码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。

可以对分页和分段机制进行配置以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）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。这种分段机制可以对段进行了分类，这样就能够限制对特定类型段的操作。

系统中所有的段都在处理器的线性地址空间内。只有逻辑地址（有时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。逻辑地址由段选择符和偏移量组成。段选择符是一个段的唯一标识。其中段描述符中包含了描述符表中的偏移（就像全局描述符表一样 GDT），该偏移指向叫作段描述符的一种数据结构。每个段有一个段描述符。段描述符描述了一个段的各种属性，比如段的大小，访问权限，段的优先权，段的类型以及该段的第一个字节在线性地址空间的位置（也称为段基址）等。通过将逻辑地址中的偏移量部分加上段基址就可以定位这个地址在段中的字节位置。段基址加偏移量就构成了处理器线性地址空间的线性地址。

address space.

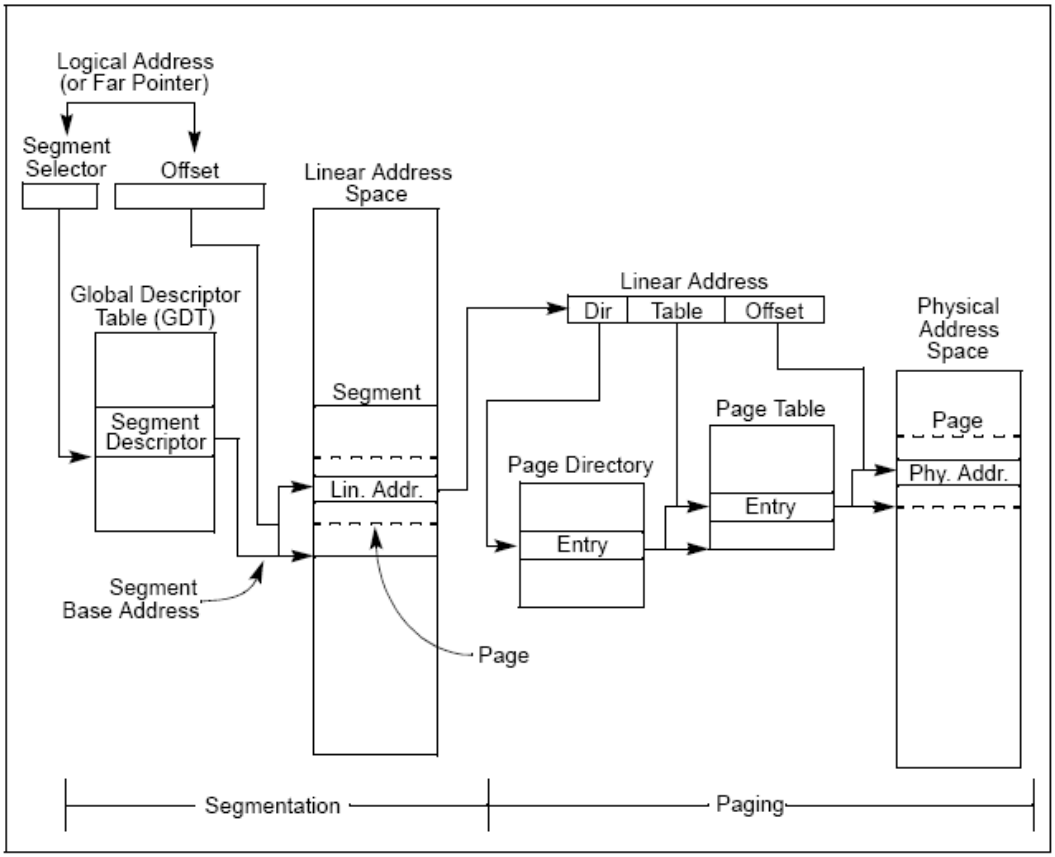

Figure 3-1. Segmentation and Paging

如果系统没有采用分页机制，线性地址空间就可以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。

分页机制支持一种虚拟内存环境。虚拟内存通过一个较小的物理内存（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。操作系统会维护一个页目录和一组页表来跟踪这些页。当一个进程试图访问线性地址空间的一个地址时，处理器通过页目录和页表将线性地址转换成物理地址，然后对其执行相应的操作（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（产生一个缺页异常）。之后，操作系统会从磁盘上读取这个页到内存中，接着执行这个程序。

当操作系统实现分页管理后，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 位的 intel 处理器所写的程序，运行在虚拟 8086 模式下也可以被透明地分页的。

3.2 段的使用

intel 架构所支持的分段机制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。

以下给出了几个例子，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 model

对一个系统而言，最简单的内存模型就是基本的平坦模型。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 intel 架构的分段机制。

在 intel 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，有同样的段限长 4GB。当段限长设置为 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 0xfffffff0 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。

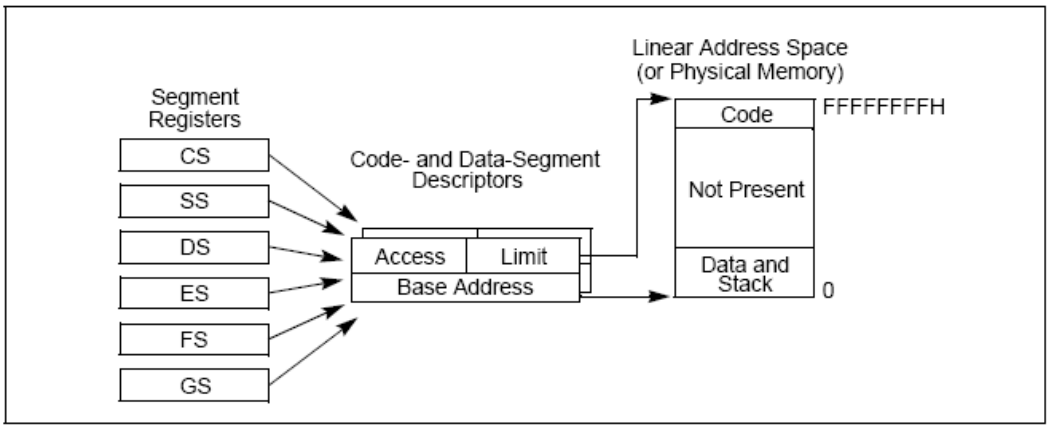

Figure 3-2. Flat Model

3.2.2 受保护的平坦模型

受保护的平坦模型与基本平坦模型类似，只是段限长被设定为在实际物理内存范围内（详

细请参考图 3-3)。如果试图访问实际内存范围以外的地址，会产生一个通用保护异常(#GP)。这个模型稍微利用了一点硬件的保护机制来防止一些程序的错误。

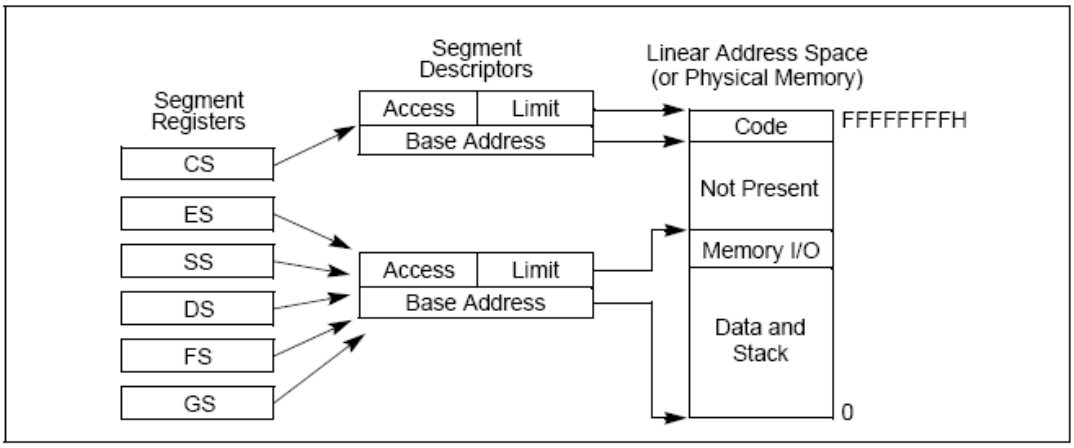

Figure 3-3. Protected Flat Model

当然，也可以使这个受保护的平坦模型更加复杂以提供更多的保护。比如，为了能在分页机制中分离普通用户和超级用户的代码和数据，需要定义 4 个段：优先权为 3（普通用户）的代码段和数据段，还有优先权为 0（超级用户）的代码段和数据段。一般来说，这些段都是互相重叠的，并且都从线性地址空间地址 0x00000000 开始。这个平坦分段模型加上一个简单的分页结构就可以在操作系统和应用程序之间起到保护作用。而且,如果为每一个进程都分配一个页结构，这样就可以在应用程序之间起到保护作用。

3.2.3多段模型

多段模型（如图 3－4 所示），充分利用了分段机制，提供了对代码，数据结构，以及程序的硬件级的强制保护。在这里，每个进程（或者任务）都被分配了自己的段描述符表以及自己的段。进程可以完全独自拥有这些分配到的段，也可以与其他进程共享这些段。单独的程序对段和执行环境的访问由硬件控制。

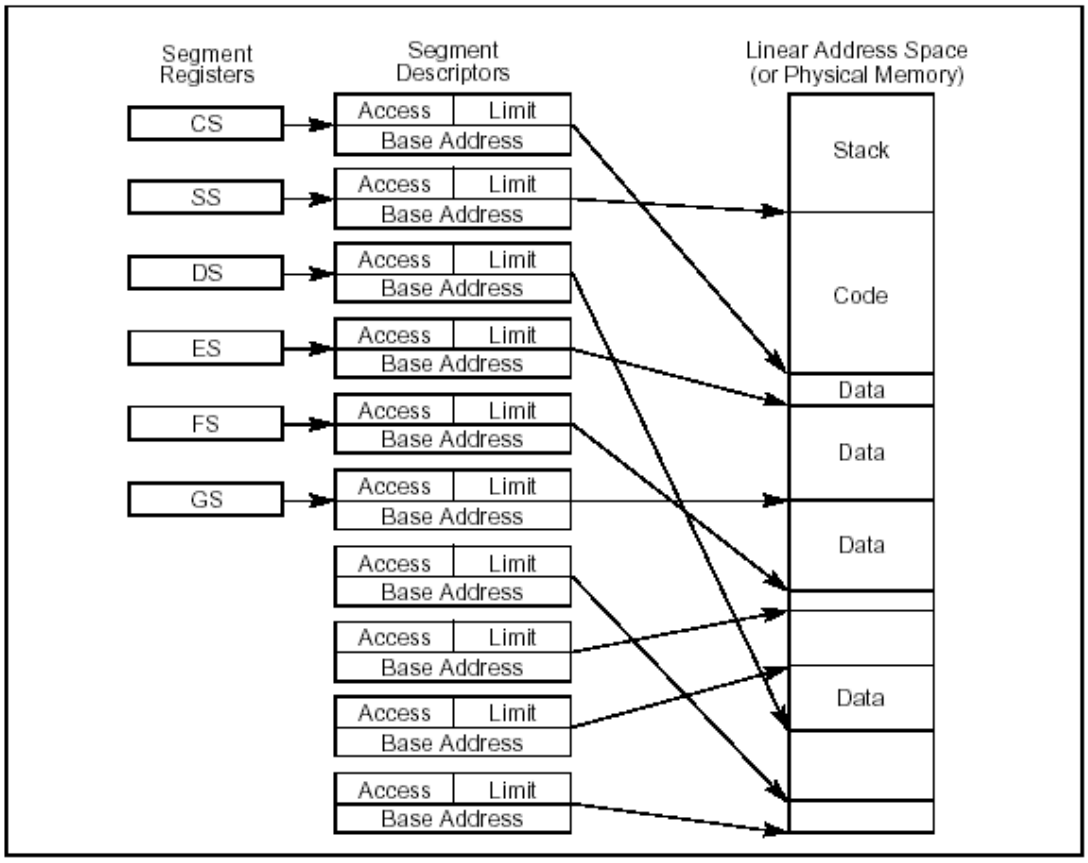

Figure 3-4. Multisegment Model

访问检查机制不仅可以避免对段限长之外的地址进行寻址，也可以避免对特定段执行非法操作。比如，因为代码段被指定为只读的，所以由硬件来防止对代码段写入数据。为段而产生的访问权限信息也可以被用来建立保护级别。保护级别也可以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

分页机制可以与图 3－2，3－3，3－4 所描述的任何一种段模型配合使用。处理器的分页机制把线性地址空间分成很多页（如图 3－1 所示）.这些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层次的保护措施，这些措施可以与段保护措施配合 i 或者取代段的保护措施。分页机制还可以提供两级保护即用户级和管理级的保护，这些保护也可以通过页偏移来指定。

3.3 物理地址空间

在保护模式下，intel架构提供最大 4GB（2^{32}字节s）的物理地址空间。这是处理器能够在地址总线上寻址的范围。这个地址空间是平坦的（未分段的），范围从 0x00000000 到

0xFFFFFFFF。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措施可以将物理内存分割成段或者页。

（在Pentium Pro处理器采用）Intel架构现在也支持物理内存空间扩展至 2^{36}字节s（64GB），其最大物理地址为FFFFFFFFFH。这个扩展由两种方式来完成：

- 使用物理地址扩展（PAE）标记来控制，这个标记是控制寄存器 CR4 的位 5
- 使用 36 位页尺寸扩展（PSE－36）特征(在奔腾 3 处理器中引入这个特征)。

有关更多 36 位物理地址寻址的信息，请参考 3.8 节，“使用 PAE 分页机制进行 36 位 物理地址寻址”和 3.9 节 “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

在保护模式下的系统架构，处理器分两步进行地址转换以最后得到物理地址：逻辑地址转换机制和线性地址空间的分页机制。

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 位的段选择符和一个 32 位的偏移量组成（参考图 3－5）。段选择符确定该字节位于哪个段，偏移量确定这个字节相对于段基址在这个段中的位置。

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。线性地址与物理地址一样，是平坦的（不分段的），空间大小为 2^{32}字节，从地址 00000000H 到FFFFFFFFH。线性地址空间包含了所有的段以及为系统而定义的各种系统表。

处理器通过如下几个步骤将逻辑地址转换为线性地址：

1. 通过段选择符中的偏移量，在 GDT 或者 LDT 中定位该段的段描述符。（仅当一个新的段选择符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以确保该段是可访问的，偏移量是在段限长范围内的。
3. 将段描述符中的段基址与偏移量相加以构成线性地址。

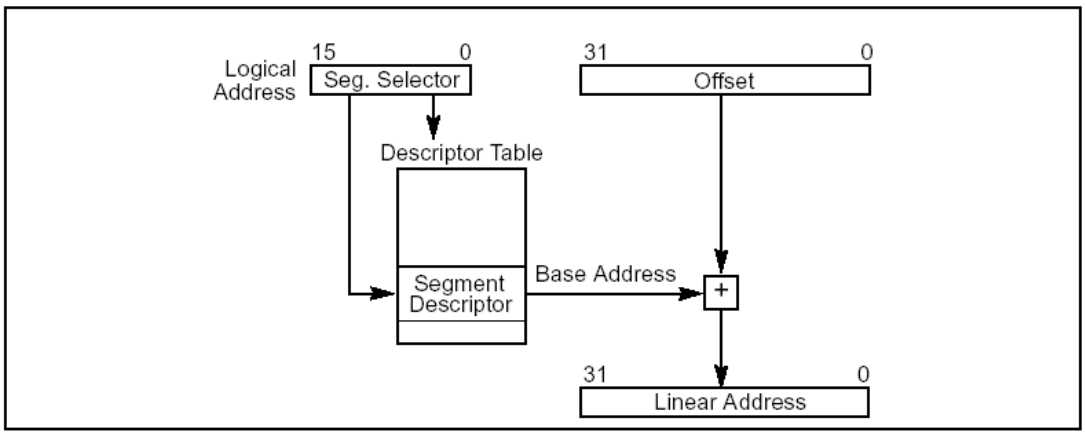

Figure 3-5. Logical Address to Linear Address Translation

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，这个线性地址可以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，就需要再一次进行地址转换，将线性地址转换为物理地址。页变换的描述请参照 3.6 节：“分页（虚拟内存）概略“。

3.4.1 段选择符

段选择符是一个 16 位的段标识符（请参照图 3－6）。它并不直接指向该段，而是指向定义该段的段描述符。一个段选择符包含以下项目：

Index （位 3～15）。选中 GDT 或 LDT 中 8192 个描述符中的某个描述符。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（基地址在 GDTR 或者 LDTR 寄存器中）。

TI(table indicator)标记

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

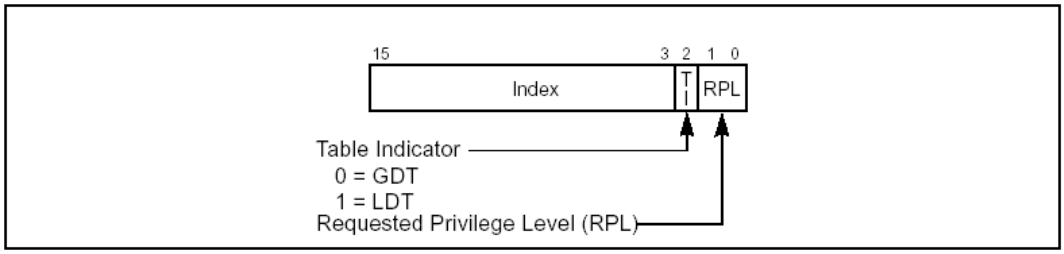

Figure 3-6. Segment Selector

请求的特权级（RPL）

（位 0 和 1）。确定该选择符的特权级。特权级从 0－3，0 为最高特权级。有关任务的 RPL 与 CPL 之间关系的描述以及该段选择符所指向的描述符的描述符

特权级（DPL），请参考第四章 4.5 节“特权级“。

GDT 中的第一项是不用的。指向 GDT 中第一项的段选择符（即选择符中的索引为 0，TI 标记置为 0）被视为空（null）段选择符。当段寄存器（CS 或 DS）被赋值为空选择符时，处理器并不产生一个异常。然而当使用值为空选择符的寄存器来访问内存时，处理器会产生一个异常。空选择符可以用来初始化未使用的段寄存器。对 CS 或者 SS 赋予一个空选择符会导致处理器产生一个通用保护异常（#GP)。

对应用程序而言，段选择符作为指针变量的一部分，是可见的。但是其值由连接程序赋予或者更改，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代码复杂度，处理器提供了 6 个段寄存器来保存段选择符（具体请参见图 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）赋予有效的段选择符。此外，处理器还提供了另外 3 个数据段寄存器（ES，FS 和 GS）供进程使用。

当一个进程要访问某个段的时候，这个段的段选择符必须被赋值到某一个段寄存器中。因此，尽管系统定义了数千个段，只有 6 个段是可以被直接使用的。其他段只有在他们的段选择符被置入这些寄存器中时才可以被使用。

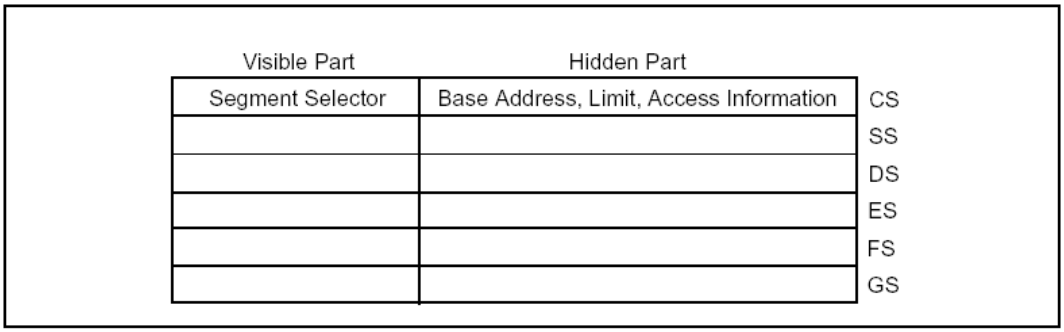

Figure 3-7. Segment Registers

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“描述符缓存”或者“影子寄存器”）。当段选择符被加载到一个段寄存器的可见部分时，处理器也通过段选择符所指向的段描述符获取了这个段寄存器的不可见信息：段基址，段限长和访问权限。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址和段限长。在一个允许多个进程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，当存储位置信息（its memory-resident version）发生变化后，用到的将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，如远指针版的CALL，JMP，RET指令，SYSENTER和SYSEXIT指令，还有IRET，INTn，INTO和INT3指令。伴随这些指令的操作，他们改变了CS寄存器的内容，有时也会改变其他段寄存器的内容。

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构造的，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。

段描述符中的标志和字段如下：

段限长字段：

指定了段的大小。处理器将这两个段限长域组合成一个20位的段限长值。根据标志位G（粒度）的不同，处理器按两种不同的方式处理段限长：

- 若G标志位为0，则该段大小可以从1字节到1M字节，段长增量单位为字节
- 若G标志位为1，则该段大小可以从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”，处理器对段限长做不同的处理。更多段类型的内容请参考3.4.3.1节“代码和数据段描述符类型”。在向上扩展段中，逻辑地址中的偏移量范围从0到段限长。超过段限长的偏移量会导致#GP异常。在向下扩展段，段限长的作用正好相反；偏移量的范围从段限长到FFFFFFFFH或者FFFFH，最大偏移量到底是FFFFFFFFH还是FFFFH，取决于B标志位的值，小于段限长的偏移量会导致GP异常。减少向下扩展段的段限长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。IA32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。

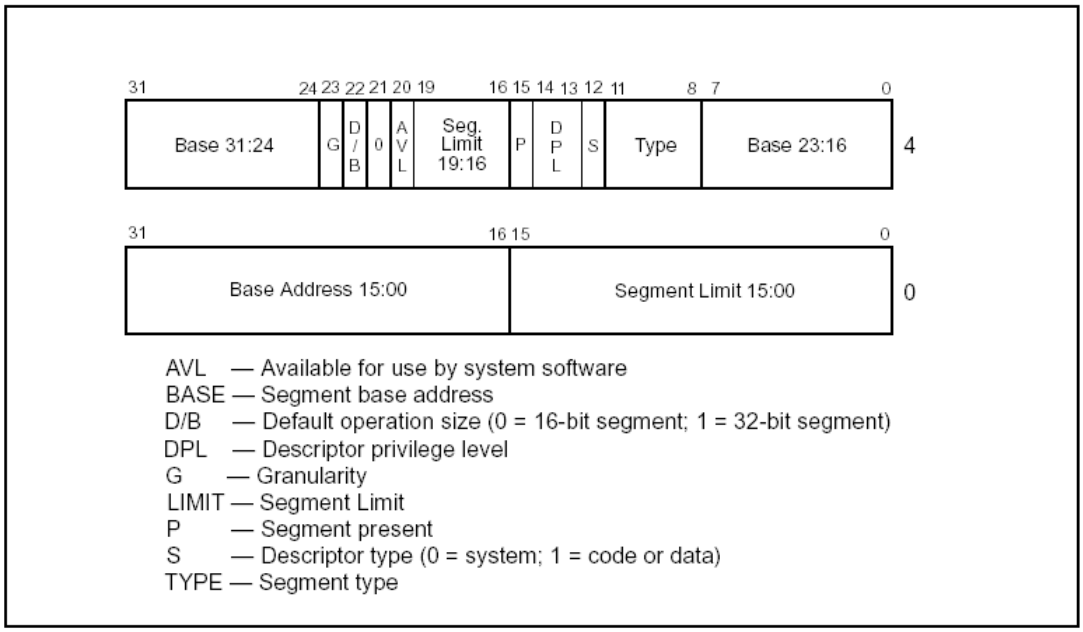

Figure 3-8. Segment Descriptor

基地址域

确定该段的第0字节在4GB线性地址空间中的位置。处理器将则3个基地址域组合在一起构成了一个32位地址值。段基址应当是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的性能最大化。

类型域

指明段或者门的类型，确定段的范围权限和增长方向。如何解释这个域，取决于该描述符是应用描述符（代码或数据）还是系统描述符，这由描述符类型标志（S 标记）所确定。代码段，数据段和系统段对类型域有不同的意义。

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

指明该段的特权级。特权级从0～3，0为最高特权级。DPL用来控制对该段的访问。关于代码段的DPL与CPL关系以及段选择符的RPL，请参考第4章*保护模式*中的4.5节”特权级“。

P（段存在）标志

标志指出该段当前是否在内存中（1表示在内存中，0表示不在）。当指向该段描述符的段选择符装载人段寄存器时，如果这个标志为0，处理器会产生一个段不存在异常（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，这样对于管理虚拟内存而言，除了分页机制还提供了另一种控制方法。

D/B(**默认操作数大小/默认栈指针大小和/或上限**)**标志**

根据这个段描述符所指的是一个可执行代码段，一个向下扩展的数据段还是一个堆栈段，这个标志完成不同的功能。（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，它指明该段中的指令所涉及的有效地址值的缺省位位数和操作符的缺省位位数。如果该标志为1，缺省为32位的地址，32位或者8位的操作符；若为0，缺省为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。用前缀67H来指定地址值长度。
- **堆栈段（由SS寄存器所指向的数据段）** 这个标志被称为B（big）标志，它为隐含的栈操作（如push，pop和call）确定栈指针值的位位数。如果该标志为1，则使用的是32位的栈指针，该指针放在32位的ESP寄存器中；若该标志为0，则使用的是放在16位 SP寄存器中的16位的栈指针。如果该堆栈段为一个向下扩展的数据段（见下一段的说明），B标志还确定了该堆栈段的地址上界。
- **向下扩展的数据段** 这个标志称为B标志，它确定了该段的地址上界。如果该标志为1，段地址上界为FFFFFFFFH（4GB）；若该标志为0，段地址上界为FFFFH（64KB）。

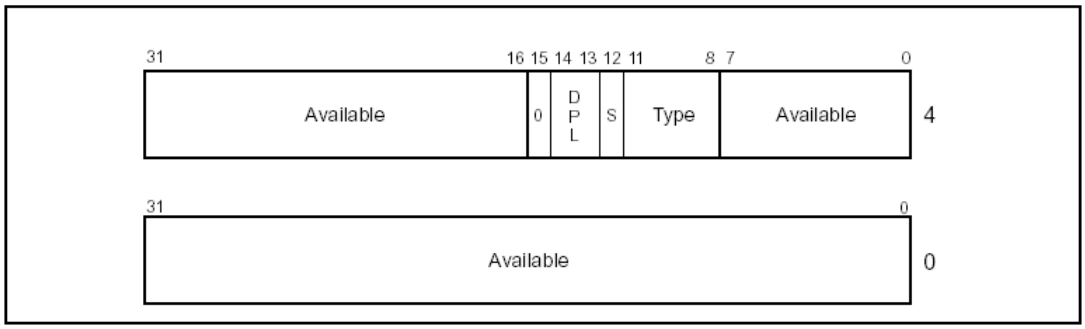

Figure 3-9. Segment Descriptor When Segment-Present Flag Is Clear

G（粒度）标志

确定段限长扩展的增量。当G标志为0，段限长以字节为单位；G标志为1，段限长以4KB为单位。（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，那么当检测偏移量是否超越段限长时，不用测试偏移量的低12位。例如，如果G标志为1，0段限长意味着有效偏移量为从0到4095。

可用及保留的位s

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

3.4.3.1.代码和数据段类型的描述符

当段描述符中的S标志（描述符类型）为1时，该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）或者代码段描述符（为1）。

对于数据段而言，描述符的类型域的低3位（位8，9，10）被解释为访问控制（A），是否可写（W），扩展方向（E）。参考表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，这取决与“是否可写”标志。

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

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

堆栈段必须是可读写的数据段。将一个不可写的数据段选择符置入SS寄存器会导致通用保护异常（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，动态改变段限长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（access 位）表示访问位自最后一次被操作系统清零后，该段是否被访问过。每当处理器将该段的段选择符置入某个段寄存器时，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debug。

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”。当有常量或者其他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符置入数据段寄存器（DS，ES，FS或者GS

寄存器）。在保护模式中，代码段是不可写的。

代码段可以是一致的，也可以是不一致的.进程的执行转入一个具有更高特权级的一致段可以使代码在当前特权级继续运行。除非使用了调用门或者任务门，进程将转入一个不同特权级的非一致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代码段的信息，请参看 4.8.1节 “直接调用或跳转到代码段”)。不访问受保护的程序和某类异常处理程序（比如除法错或者溢出）的系统程序可以被载入一致的代码段。不能被更低特权级的进程访问的程序应该被载入非一致的代码段。

注意：

无论目标段是否为一致代码段，进程都不能因为call或jump而转入一个低特权级（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通用保护异常（GP）.

所有的数据段都是非一致的，这就意味着数据段不能被更低特权级的进程访问（特权值较大的执行代码）。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或者一个LDT中的段描述符在ROM中，当程序或者处理器试图更改在ROM中的段描述符时，处理器将进入一个无限循环。为了防止此类问题的发生，可以将所有在ROM中的段描述符的访问位置位。同时，除去所有操作系统代码中试图更改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，该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“，它们或者持有指向在代码段的过程的入口点

的指针，或者持有TSS（任务门）的段选择符。表3－2显示了对系统段描述符和门描述的类型域的译码

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

Type Field					Description
Decimal	11	10	9	8	
0	0	0	0	0	Reserved
1	0	0	0	1	16-Bit TSS (Available)
2	0	0	1	0	LDT
3	0	0	1	1	16-Bit TSS (Busy)
4	0	1	0	0	16-Bit Call Gate
5	0	1	0	1	Task Gate
6	0	1	1	0	16-Bit Interrupt Gate
7	0	1	1	1	16-Bit Trap Gate
8	1	0	0	0	Reserved
9	1	0	0	1	32-Bit TSS (Available)
10	1	0	1	0	Reserved
11	1	0	1	1	32-Bit TSS (Busy)
12	1	1	0	0	32-Bit Call Gate
13	1	1	0	1	Reserved
14	1	1	1	0	32-Bit Interrupt Gate
15	1	1	1	1	32-Bit Trap Gate

更多系统段描述符的信息，请参照3.5.1节“段描述符表“和第六章 *任务管理* 6.2.2节”TSS 描述符“。更多关于门描述符的信息，请参考第四章 *保护模式* 中4.8.2节“门描述符“；第五章 *中断和异常处理* 中5.9节 “IDT 描述符”；第六章 *任务管理* 中6.2.4节“任务门描述符”。

3. 5. 1段描述符表

一个段描述符表是一个段描述符的数组（参看图3－10）。段描述符表的长度不固定，可以最多包含8192（2^{13}）个8字节的描述符。有两种描述符表“

- 全局描述符表（GDT）
- 局部描述符表（LDT）

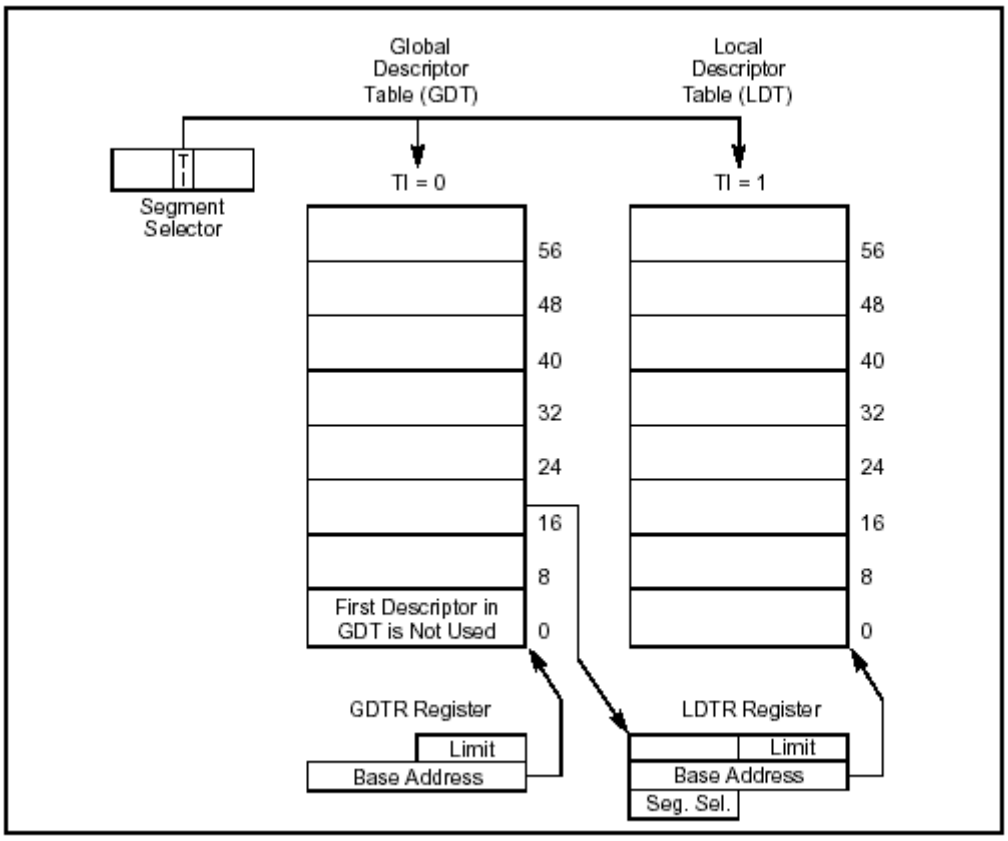

Figure 3-10. Global and Local Descriptor Tables

系统必须定义一个GDT，以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的线性基地址和限长必须被装载入GDTR寄存器（请参考第二章 *系统架构概况* 中2.4节“内存管理寄存器”）。GDT的基址应当按照8字节方式对齐，这样可以获得最好的处理器性能。GDT的限长值是按比特计算的。在段中，段限长加上段基址可以用获取段中最后一个比特的有效地址。0限长值表示只有一个有效的比特。因为段描述符总是8比特长，GDT的限长应该总是八的整数倍减一（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描述符的段选择符被装载入数据段寄存器（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通用保护异常（GP）。使用这个指向NULL描述符的段选择符来初始化段寄存器，这样可以确保在不经意地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符，都要在GDT中有一个段描述符。LDT

的段描述符可以位于GDT中的的任何地方。关于LDT段描述符类型的信息，参考3.5节 “系统描述符类型”。

LDT是通过它的段选择符来访问的。为了避免在访问LDT时进行地址翻译，LDT的段选择符，线性基地址，段限长和访问权限都放在LDTR寄存器中。（参考第二章 系统架构概况中的2.4节 “内存管理寄存器”）。

LDT是通过它的段选择符来访问的。为了消除在访问LDT时的地址转换，LDT的段选择符，线性基址，段限长和访问权限都存放在LDTR寄存器中（请参见第二章 系统架构概况中的2.4节 “内存管理寄存器”）。

当GDTR寄存器被装载时（使用SGDT指令），一个48位的伪描述符也被放入内存中（参看图3－11）。为了避免在用户模式下（特权级为3）发生对齐检查错误，伪描述符应该放在一个奇数字地址上（即，该地址对4取模的结果为2）。这样可以使处理器存一个对齐的字，后面紧跟着一个对齐的双字。用户模式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错误。在使用SIDT指令装载IDTR寄存器的时候，也应该使用同样的对齐方法。当装载LDTR或者任务寄存器时（分别使用SLTR和STR指令），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）。

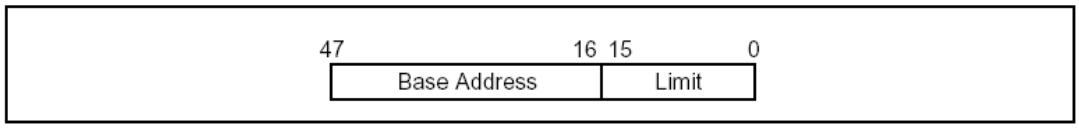

Figure 3-11. Pseudo-Descriptor Format

3．6 分页（虚拟内存）

当操作系统在保护模式下时，intel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）映射到一个较小的内存和磁盘存储空间。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（通常一页是4KB），这些页可以被映射到物理内存或者磁盘存储空间。当一个进程（或者任务）引用一个内存中的逻辑地址的时候，处理器将这个地址转换为线性地址，然后使用分页机制将线性地址转换为相应的物理地址。如果包含该线性地址的页不在内存中，处理器产生一个缺页异常（PF）。典型的缺页异常处理程序指导操作系统将该页从磁盘上调入内存（在这个过程当中，有可能另外一个页被调出内存，存放到磁盘上）。当该页被调入内存后，导致缺页异常产生的

指令将被重新执行。处理器用来映射线性地址到物理地址的信息和产生缺页异常（如有必要）的信息都在页目录和页表中，页目录和页表都存放在内存中。

分页与分段不同，它使用固定大小的页面。与段不同，页有固定的尺寸，段的大小与它所持有的代码、数据结构总和的大小。如果仅仅使用分段作为唯一的地址转换形式，一个数据结构必须全部在物理内存中。但是如果启用了分页，一个数据结构可以部分在内存，部分在磁盘。

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和页表项都被缓存在一个叫做转换后备缓冲区（translation lookaside buffers，TLBs）的设备中。TLBs可以满足多数的读当前页目录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TLBs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TLBs的更多内容，请参照3.11节 转换后备缓冲区(TLBs)

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寄存器的位31（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寄存器的位4（在Pentium和Pentium Pro处理器中引入）
- PAE（物理地址扩展）标志，CR4寄存器的位5（在Pentium Pro处理器中引入）

PG标志可以启用分页机制。通常操作系统在处理器初始化的时候设置这个标志。如果处理器的分页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标志允许系统使用具有更大尺寸的页：4MB的页或者2MB的页（当设置PAE标志的时候）。当PSE标志被清零的时候，则使用通常的4KB的页。有关PSE标志的更多使用信息，请参考3.7.2节 *线性地址转换（4MB 页）*，3.8.2节 *使用扩展寻址的线性地址转换* （2MB或者4MB的页）和3.9节 “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。仅当分页机制被启用后，才可以使用物理地址扩展。它依赖页目录和页表来寻址超过FFFFFFFFH的地址。更多关于使用PAE来进行物理地址扩展的信息，请参考3.8节 *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替代扩展36位物理寻址的方法。这种分页机制使用页尺寸扩展模式，并且更改页目录项来寻址物理地址在FFFFFFFFH以上的内存。PSE

－36标志（当用源操作数1来执行CPUID指令以后，EDX寄存器的位17）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，请参考3.9节 “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

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都包含在4个数据结构中：

- 页目录－一个由32位页目录项（page－directory entries：PDEs）组成的数组。它被放在一个4KB的页中。页目录最多包含1024个页目录项。
- 页表－ 一个由32位页表项（page－table entries：PTEs）组成的数组。它存放于一个4KB的页中。页表最多包含1024个页表项。（对于2MB或者4MB的页，不使用页表。这些页直接从一个或者更多的页目录项映射。
- 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。仅当启用物理地址扩展时才使用这个数据结构（参考3.8节 *物理地址扩展*）

这些表可以用来在常规的32位物理地址寻址时访问4KB或者4MB的页，也可以用来在寻址扩展的(36位)物理地址时访问4KB，2MB或者4MB的页。表3－3显示了通过分页控制标志的各种不同设置与PSE－36 CPUID指令得到的标志所获得的页的大小和物理地址的大小。每个页目录表项都包含了一个PS（page size）标志，这个标志说明了这个页目录表项所指向的是一个页表（其每个表项指向一个4KB的页）（PS置为0），或者这个页目录表项直接指向一个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

以下部分描述了在使用32位物理地址，最大物理地址空间为4GB时，IA－32架构的页变换机制。3.8节 “使用PAE分页机制时的36位物理寻址” 和3.9节 “使用PSE－36分页机制的36位物理寻址”描述了这种页变换机制的扩展,这种扩展支持36位物理地址,可支持最大64GB物理地址空间。

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展示了在映射线性地址到4KB的页时，页目录和页表的层次结构。页目录中的表项指向页表，而页表的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}的页，其跨越的线性地址空间为2^{32}字节（4GB）。

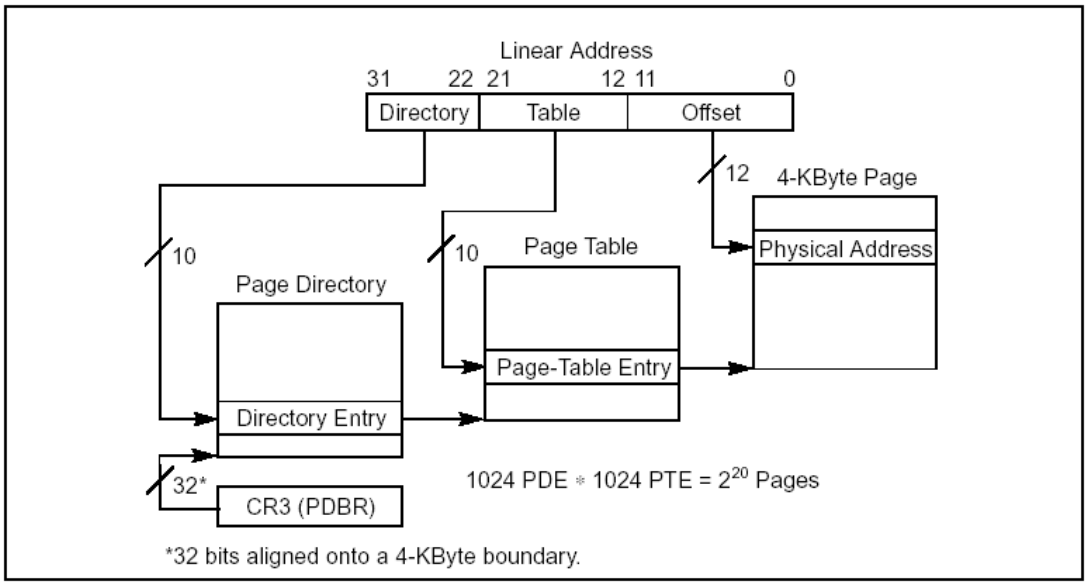

Figure 3-12. Linear Address Translation (4-KByte Pages)

为了选择不同的表的入口，线性地址被分为3个部分：

- 页目录表项－位 22到位31作为一个表项在页目录中的偏移量。该表项提供了一个页表的物理基地址。
- 页表项－线性地址位12到位21提供了一个表项在所选的页表中的偏移量。该表项提供了物理内存页的物理基地址。
- 页偏移量－位0到位11提供了该地址在页中的偏移量。

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

3.7.2.线性地址转换（4MB页）

图3－12显示如何使用页目录来映射线性地址到4MB的页。该页目录的表项指向物理内存中的4MB的页。这种分页方法可以将1024个页映射到4GB的线性地址空间。

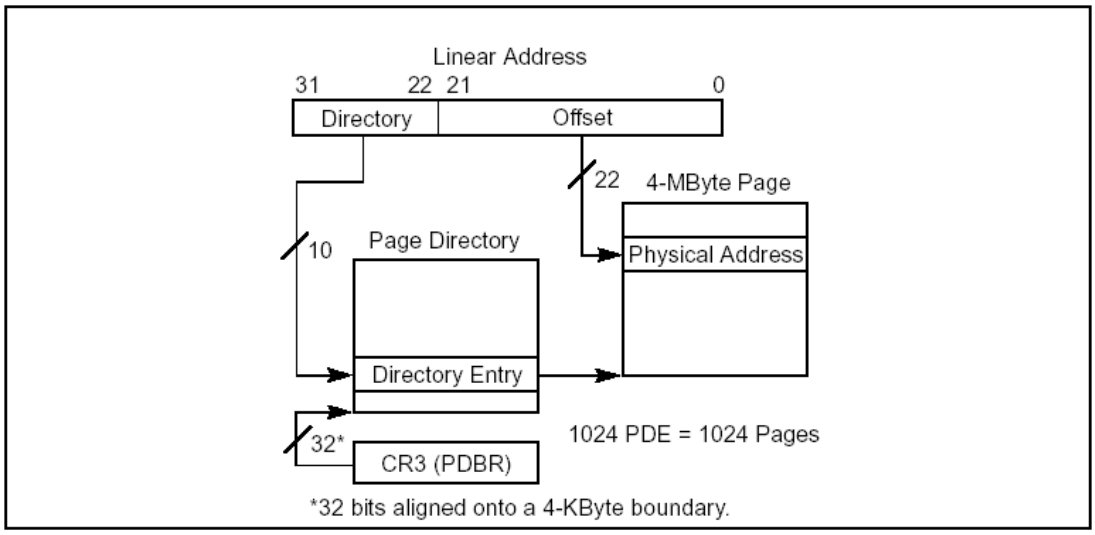

Figure 3-13. Linear Address Translation (4-MByte Pages)

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（page size PS）标志（参照图3－14），可以启用4MB的页。通过设置这些标志，线性地址被分为2部分：

- 页目录表项－线性地址的22位到31位提供了该表项在页目录中的偏移量。该表项提供了4MB的页的物理基地址。
- 页偏移量－线性地址的0位位到21位位提供了该地址在页中的偏移量。

注意

（仅针对奔腾处理器）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，TLBs必须被刷新。否则可能由于处理器使用了存在TLBs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）TLBs，请参考第九章 *处理器缓存控制* 10.9节 使TLBs失效。

3.7.3.混合使用4KB和4MB的页

当设置CR4中的PSE标志时，可以通过同一个页目录来访问4MB的页和4KB的页的页表。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内核放在大尺寸的页中来减少TLB失效（TLB Miss），从而提高了整体系统性能。处理器在不同的的TLB中维护4MB的页表项和4KB的页表项。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

通过将两个页目录表项指向一个普通的页表，IA－32架构可以使用内存别名。需要通过这种方式来实现内存别名的软件必须处理好页目录项和页表项中“访问位”和“脏位”的一致性。如果两个页目录项的访问位和脏位不一致，将会导致处理器死锁。

3.7.5.页目录基地址

当前页目录的物理地址存放在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，请参考第二章 *系统架构概略* 图2－5及2.5节 “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来显式的改变PDBR的值，也可以在任务切换时，隐含的改变它。（关于如何为任务设置CR3，请参考第6章 *任务管理* 6.2.1节 “任务状态段（TSS）”）。

在PDBR中没有为页目录而设的存在标志。当一个任务被挂起时，与之相关的页目录也许不在内存。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镜像所指的页目录必须已经在内存中。只要任务还是处于active状态，该页目录就必须保留在内存中。

3.7.6. 页目录项和页表项

图3－14显示了系统使用4KB的页和32位的物理地址时，页目录项和页表项的格式。该图还显示了当系统使用4MB页和32位的物理地址时页目录项的格式。图3－14和图3－15所示的各个表项中的标志和域的功能阐述如下：

页基址 位12到位22

（页表项，指向4KB的页）确定了一个4KB页的第一个字节的物理地址。这个域被解释为该物理地址的高20位，这就强迫页都是4KB对齐的。

（页目录项，指向4KB页的页表）确定了一个页表的第一个字节的物理地址。这个域被解释为该物理地址的高20位，这就要求页表的基地址是4KB对齐。

（页目录项，指向4MB的页）确定一个4MB页的第一个字节的物理地址。在这种情况下，这个域只有位22到位31是用到的（从Intel架构的Pentium II处理器开始，位12到位21被保留并且被置为0）。这个基地址被解释为这个物理地址的高10位，这要求4MB的页必须是4MB对齐的。

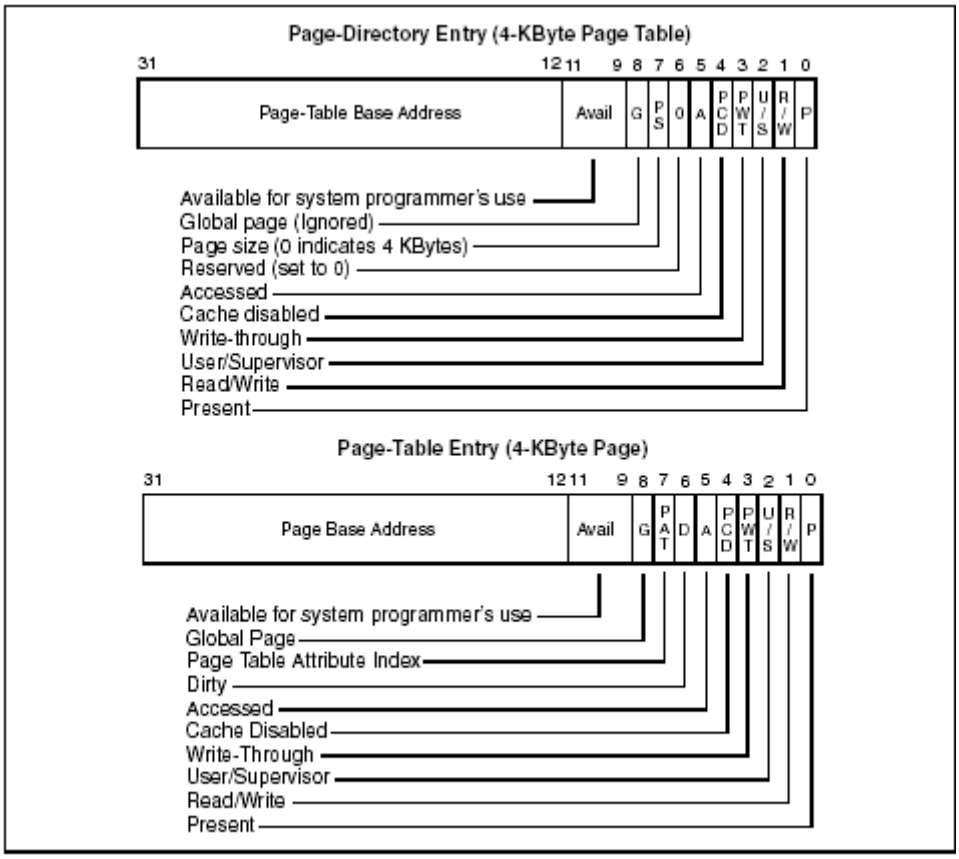

Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

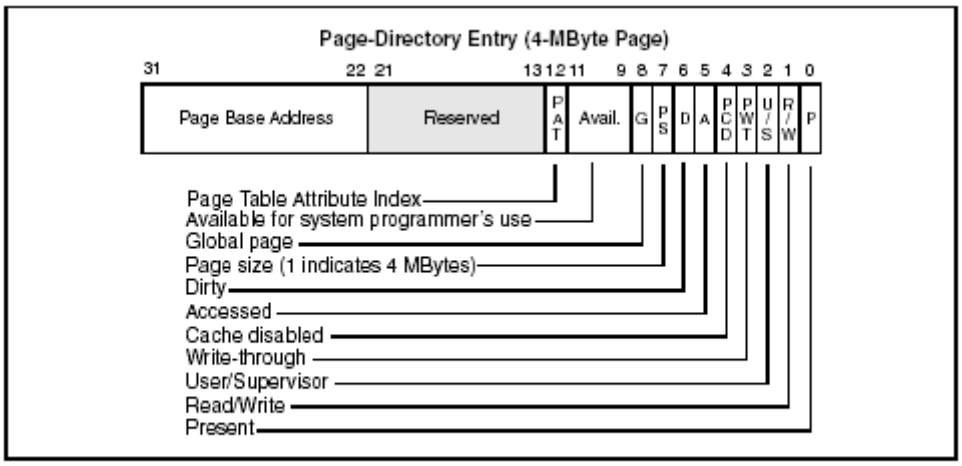

Figure 3-15. Format of Page-Directory Entries for 4-MByte Pages and 32-Bit Addresses

存在（P）标志，位0

该标志表明，该表项所指向的页或者页表当前是否在内存中。当置位该标志时，这个页在物理内存中，将执行地址转换。当该标志清零时，表示这个页不在内存中，如果处理器试图访问该页，将产生一个缺页异常（PF）。

处理器并不置位或者清零该位；而是由操作系统来维护该标志的状态。

如果处理器产生一个缺页异常，操作系统必须按序执行如下操作：

1. 如果有必要，将该页从磁盘拷贝到内存中。
2. 将该页地址装载入页表或者页目录项并设置它的存在标志。其他的位，比如脏位和访问位，也必须同时被设置。
3. 使TLB中的当前页表项失效（有关TLBs的信息以及如何使他们失效，请参考3.7节 “转换后备缓冲区 (TLBs)”）。
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，位 1

该标志确定对一个页或者一组页（比如，一个指向一个页表的页目录项）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标志共同起作用。有关使用这些标志的详细讨论，请参考第四章 *保护模式* 4.11节 “页层次的保护” 和表4－2。

普通用户/超级用户（U/S）标志，位 2

该标志确定一个页或者一组页（比如，一个指向一个页表的页目录项）的用户权限。当这个标志被清零，该页的用户权限为超级用户的权限；该标志置位时，该页的用户权限为普通用户权限。这个标志与R/W标志和CR0寄存器中的WP标志共同起作用。有关使用这些标志的详细讨论，请参考第四章 *保护模式* 4.11节 “页层次的保护” 和表4－2。

页级直写(PWT) 标记, 位 3

控制单个页（页表）的直写或者回写缓存策略。当PWT标志被置位时，启用页表的直写缓存机制，当PWT标志被清零时，回写（write－back）缓存与页或者页表关联；当CR0寄存器中的CD（cache disable）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,可参考第9章 *内存缓存控制* 10.5节“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，可参考第二章 *系统架构概要* 2.5节 “控制寄存器”。

页层次的缓存禁用（PCD）标志 位 4

控制单个页或者页表的缓存。当该标志被置位时，相关页或者页表的缓存被禁止；当该位被清零时，相关页或页表可以被缓存。这个标志可以用来禁止缓存包含内存映射I/O端口的页或者即使被缓存，也不能对性能有提高的页。当CR0寄存器中

的CD（cache disable）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，请参考第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，请参考第二章 *系统架构概略* 2.5节。

访问（A）标志，位 5

指明这个页或页表是否曾经被访问过。内存管理软件通常会在这个页或者页表被载入内存时，清零该位。当该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”标志，就是说一旦被设置，处理器不会隐式的给它清零。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，位6

指明该页是否曾经被写入过（在指向页表的页目录项中，不使用该标志）。通常，内存管理软件在该页刚被载入内存时，将该标志清零。当该页的第一次写操作完成后，处理器置位该标志。这个标志是一个粘性标志，就是说，一旦被设置，处理器不会隐式的对它清零。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，位 7

确定页的尺寸。该标志仅被用于页目录项。当该标志被清零时，页尺寸为4KB，页目录项指向一个页表。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，页的尺寸位2MB），页目录项指向一个页。如果页目录项指向一个页表，所有与那个页表相关的页都是4KB。

页属性表索引（PAT）标志，4KB页表项的位7和4MB页目录项的位12

（在奔腾III处理器中采用）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节 “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，位 8

（在奔腾Pro处理器中引入）指明全局页。当一个页被标明为全局，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他的系统代码）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来说，这个标志不起作用的。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，请参考3.7节“转换后备缓冲区 (TLBs)”（在奔腾和更早期的intel架构处理器中，该位是被保留的）。

保留的、可供软件使用的位

对于所有的IA32处理器，位9，位10和位11都是可以为软件所用。（当“存在位”被清零时，位1到位31都可以为软件使用－见图3－16）在指向页表的页目录项中，位6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错误。

对于奔腾II及早期的处理器，页表项的位7被保留，并置为0。对于4MB页的页目录项，位12到位21都是被保留的，并且应当被置为0。

对于奔腾III及后来的处理器，4MB页的页目录项中，位13到位21都是被保留的，必须被置为0。

3.7.7. 不在场的页目录和页表项

当一个页表或者页目录项的“存在（present）”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)

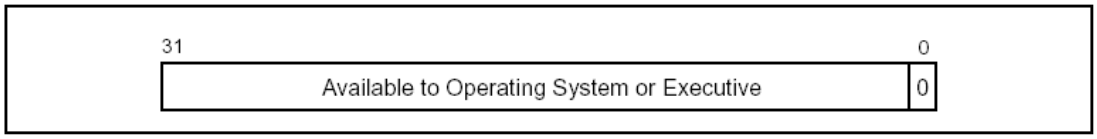

Figure 3-16. Format of a Page-Table or Page-Directory Entry for a Not-Present Page

3.8. 使用 PAE 的分页机制的 36 位物理寻址

PAE分页机制以及对36位物理寻址的支持，是在IA32架构的奔腾pro处理器中采用的。在IA32处理器中实现这个特性是通过CPUID指令的特性标志PAE（当CPUID指令的源操作数是2时，EDX寄存器的位6就是这个特性标志）。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。为了能使用这个选项，必须设置如下的标志：

- CR0寄存器中的PG标志（位 31）－开启分页
- CR4寄存器中的PAE标志（位5）置位，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，一个页目录项可以指向一个2MB的页，也可以指向一个页表，这个页表的表项指向4KB的页）。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页的页目录和页表也就可以有最多512个表项了。
- 线性地址变换的层次中，一个叫做页目录指针表的新表将被加入。这个表有4个64位的表项。在线性变换的层次中，这个表在页目录之上。随着物理地址扩展机制的开启，处理器支持4个页目录。
- 寄存器CR3（PDPR）中20位的页目录基地址被27位 的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，这就迫使页目录指针表的地址是32字节对齐的。
- 线性地址变换允许将32位的线性地址映射到更大的物理地址空间中。

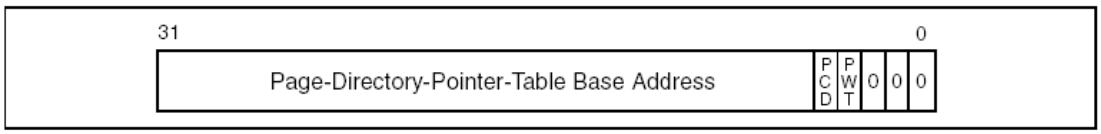

Figure 3-17. Register CR3 Format When the Physical Address Extension is Enabled

3.8.1. 开启PAE时的线性地址变换（4KB页）

图3－18显示了当启用PAE分页机制进行线性地址到4KB页映射时，页目录指针表，页目录和页表的层次结构。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－位30到位31，给出了该页目录指针表项在页目录指针表中的偏移量。被选中的表项给出了一个页目录的基地址。
- 页目录项－位21到位29，给出了在被选中的页目录中的偏移量。被选择的目录项给出了一个页表的基地址。
- 页表项－位12到位20，给出了在被选中的页表中的偏移量。被选中的页表项给出了一个页在内存中的物理基地址。
- 页偏移量－位0到位11，给出了在被选中的页中的偏移量。

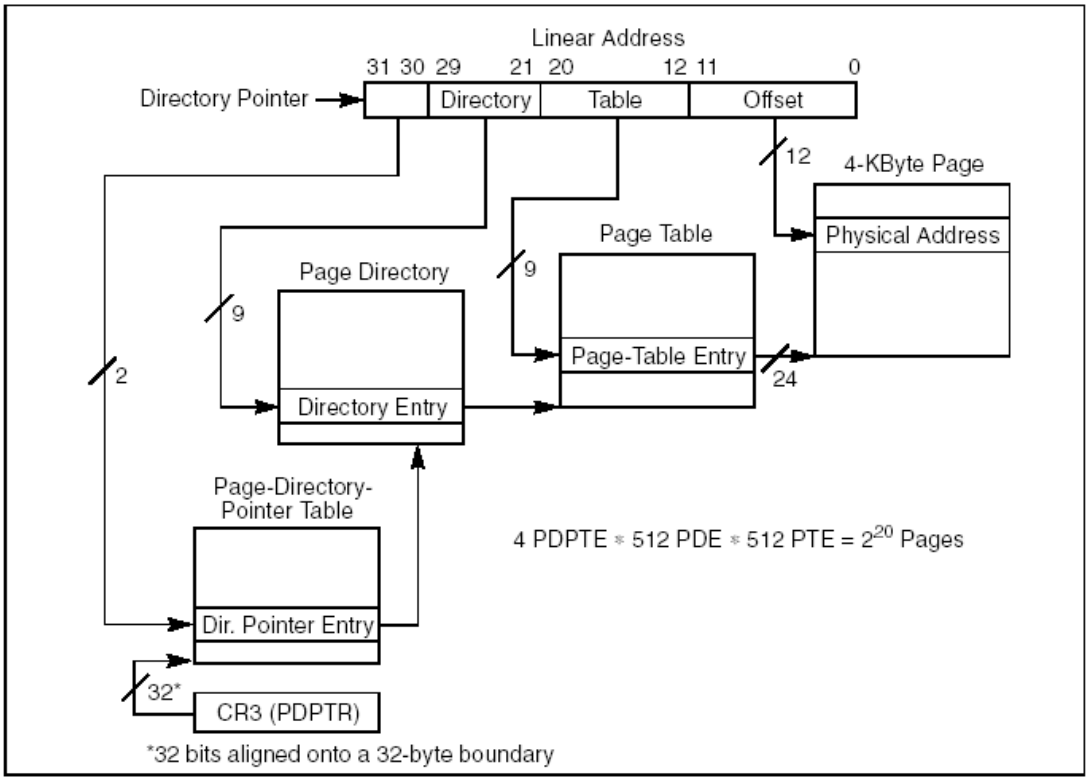

Figure 3-18. Linear Address Translation With PAE Enabled (4-KByte Pages)

3.8.2启用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将线性地址映射到2MB的页。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，通过设置页目录项中的页尺寸（PS）标志（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－位30到位31，给出了一个页目录指针表项在页目录指针表中的偏移量。该页目录指针表项给出了一个页目录的基地址。

- 页目录项－位21到位29，给出了一个页目录项在页目录中的偏移量。该页目录项给出了一个2MB页的基地址。
- 页偏移量－位0到位20，给出了该地址在页中的偏移量。

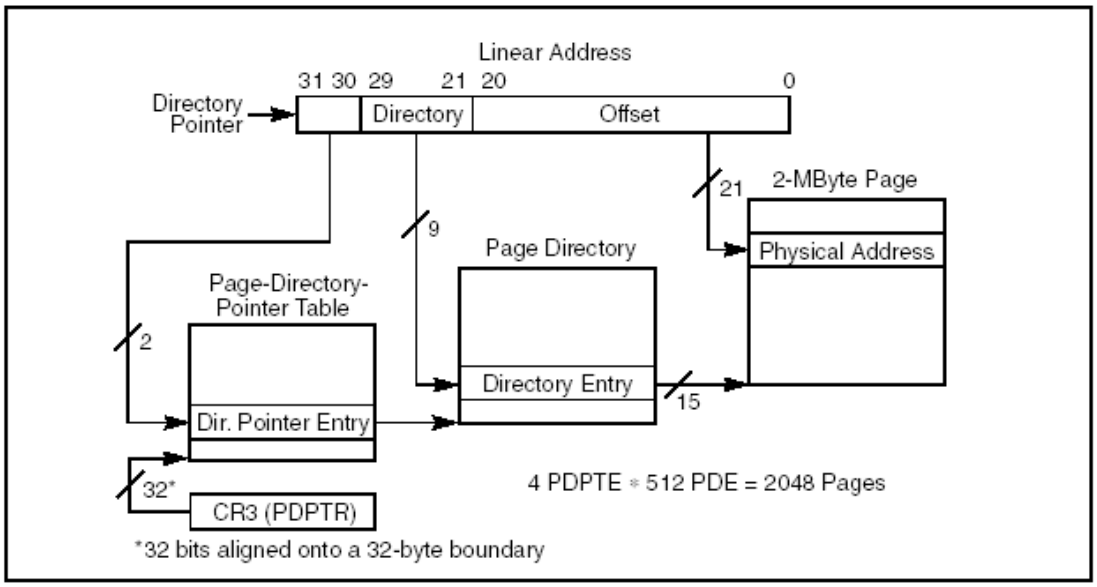

Figure 3-19. Linear Address Translation With PAE Enabled (2-MByte Pages)

3.8.3使用扩展的页表结构来访问完全扩展的物理地址空间

前两节描述的页表结构给出了64G字节的扩展物理地址空间中的4G字节的访问方法，另外的4G字节的物理内存可以通过下面两种方法中的一种进行访问：

- 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一个页目录和页表集合。
- 改变页目录指针表的表项，使其指向另外一个页目录，这个页目录又会指向另外一个页表集合。

3.8.4.启用扩展寻址后的页目录项和页表项

图3－20显示了当使用4KB页，使用了36位扩展物理地址时，页目录指针表项，页目录项和页表项的格式。图3－21当使用2MB和36位扩展物理地址时，页目录指针表项和页目录项的格式。这些表项中的标志功能与3.7.6节 “页目录项和页表项”中描述的功能是一样的，其中主要的不同之处如下：

- 增加的页目录指针表项
- 表项的大小从32位增加到了64位
- 页目录和页表的最多项数为512个
- 每个项中，物理基地址域扩展到了24位

注意

现行的实现了PAE机制的IA－32处理器在装载页目录指针表项时，使用非缓存访问。这种行为是模式特定行为，而非架构特定行为。未来的IA－32处理器也许会缓存页目录指针表项

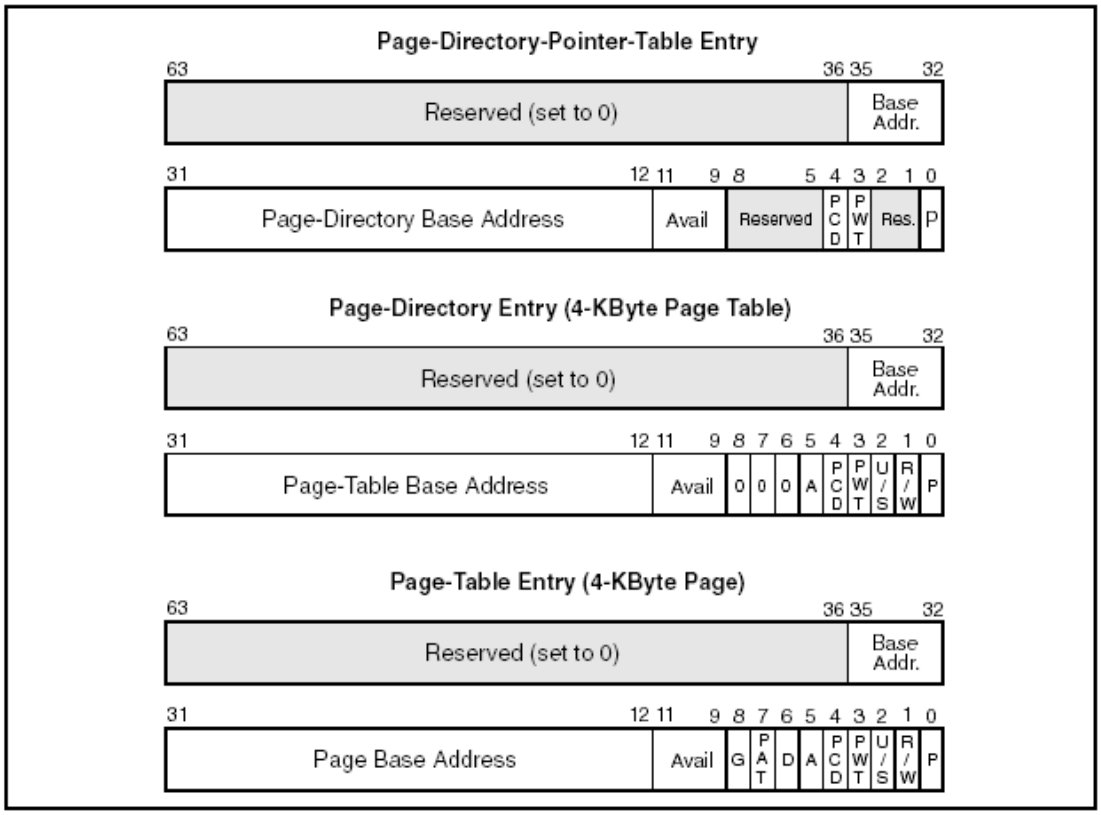

Figure 3-20. Format of Page-Directory-Pointer-Table, Page-Directory, and Page-Table Entries for 4-KByte Pages with PAE Enabled

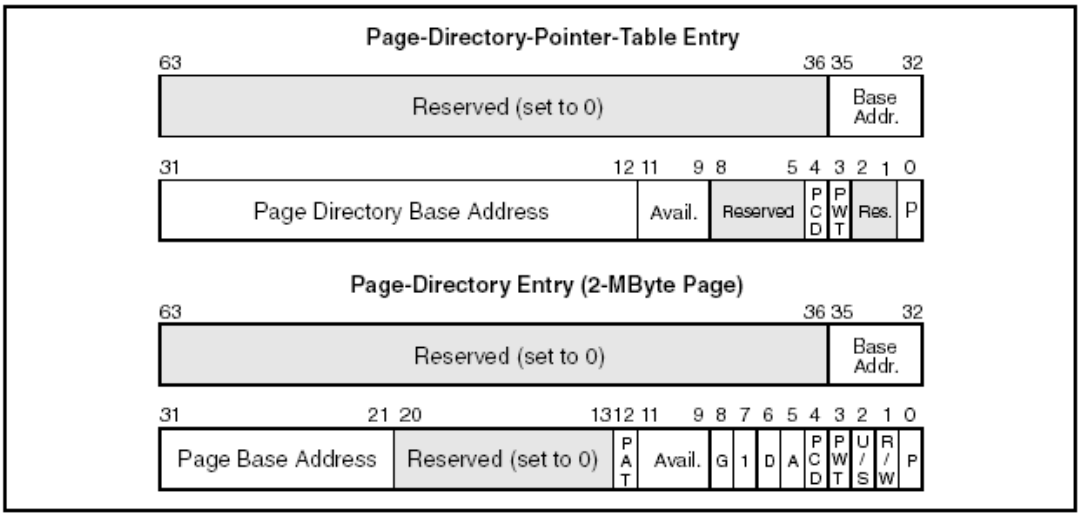

Figure 3-21. Format of Page-Directory-Pointer-Table and Page-Directory Entries for 2-MByte Pages with PAE Enabled

根据表项的不同，表项中的物理基地址的说明如下：

- 在页目录指针表项中，基地址是一个4KB页目录的第一个字节的物理地址

- 在页目录项中，基地址是一个4KB页表或2MB页的第一个字节的物理地址
- 在页表项中，基地址是一个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页的页目录项），基地址都被视为36位物理地址的高24位，这就迫使页表和页都是4KB对齐的（这样，36位物理地址的低12位都为0）。当页目录项指向一个2MB的页时，基地址被视为36位物理地址的高15位，这就迫使2MB的页都是2MB对齐的（这样，36位物理地址的低21位为0）。

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

页目录项中的页尺寸标志（位7）可以判断该表项指向一个页表还是指向一个2MB的页。当该标志被清零时，表项指向一个页表；当该标志被置位时，表项指向一个2MB的页。这个标志使得4KB和2MB的页在一个页表集合中混用。

访问标志（A）(位5)和脏标志（D）（位6）供指向页的表项使用。

所有物理地址扩展表项的位9，10和11都可为软件所用。（当“存在位”为0时，位1到位63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，而页目录项和页表项中的保留位没有被置为0，处理器产生一个页错误（＃PF）；如果页目录指针表项中的保留位没有被置为0，处理器会产生一个一般错误（＃GP）。

3.9.使用 PSE36 分页机制时的 36 位物理寻址

PSE36分页机制提供了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的页到64GB地址空间。与PAE机制一样，处理器提供了额外的4个地址线引脚来适应额外的地址位。

PSE36机制在奔腾III处理器中引入IA32架构。这种机制是否可用可以通过PSE36特征位来判断（执行源操作数为1的CPUID指令后，EDX寄存器的位17）。

如表3－3中所示，如下的标志必须被设置或者清零来启用PSE36分页机制：

- PSE36 CPUID特征标志，当置位时，表示可以使用PSE36分页机制
- CR0寄存器中的PG标志（位 31），置为1，启用分页
- CR4寄存器中的PAE标志（位 5），置为0，禁用PAE分页机制。

- CR4寄存器中的PSE标志（位 4）和页目录项中的PS标志，置为1，启用页尺寸扩展来使用4MB页。
- 或者将CR4寄存器中的PSE标志（位 4）置为1，将页目录项中的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：

- 页目录项－位22到位35，给出了一个表项在页目录中的偏移量。该表项给出了一个36位地址的高14位以确定一个4MB页的基地址
- 页偏移量－位0到位21给出了一个物理地址在页中的偏移量

这种分页方法可以将1024个页映射到64ＧＢ物理地址空间。

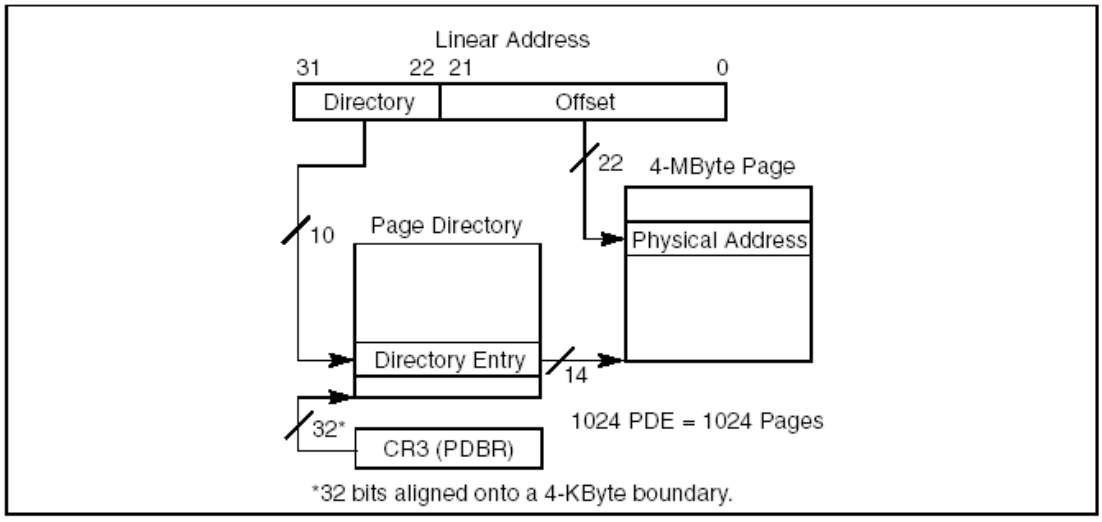

Figure 3-22. Linear Address Translation (4-MByte Pages)

图3－23显示了使用4MB页和36位物理地址时的页目录项的结构。3.7.6.节 “页目录和页表项”描述了位0到位11中的各个标志和域的功能。

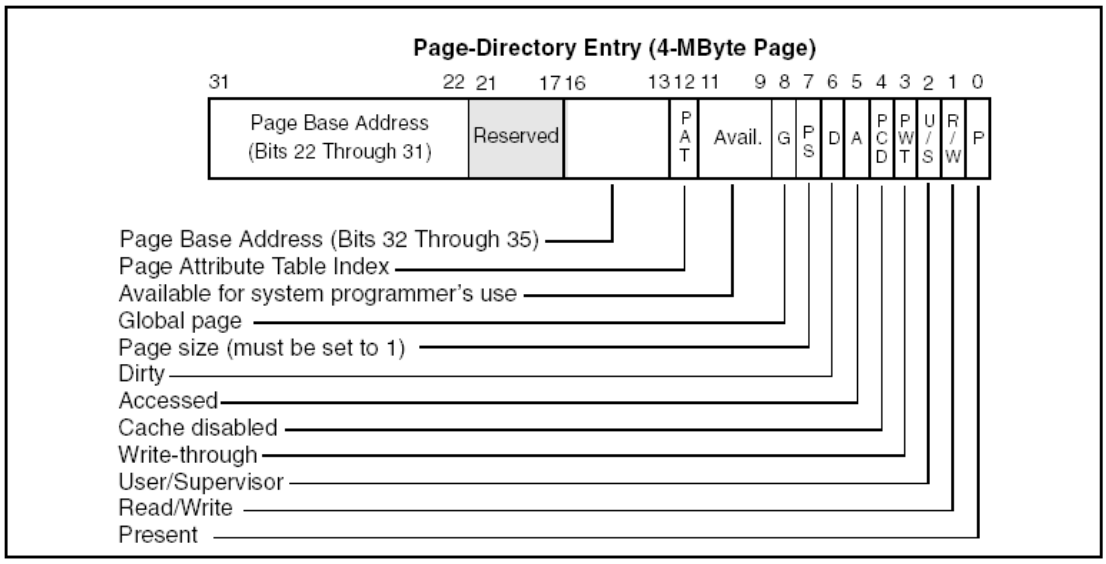

Figure 3-23. Format of Page-Directory Entries for 4-MByte Pages and 36-Bit Physical Addresses

3.10.段到页的映射

分段和分页机制为IA32架构提供了很多方法来进行内存管理。当分段和分页结合起来，可以通过几种不同的方法来将段映射到页。比如说，在平坦寻址环境中，代码，数据和堆栈模块被映射到一个或者多个段(最大为4GB)，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（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和页表集合将线性地址空间映射到虚拟内存。

段的尺寸可以比页的尺寸小。如果某个段在一个不与其他段共享的页内，该页内其他的内存就浪费了。一个小的数据结构，比如说，一个一字节的信号量，如果它将它自己放在一个页内，它就占据了4KB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。

IA32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，一个段可以包含一个页的结尾和另外一个页的开始。

如图3－24中所示，让每个段都有其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中有一个单独的表项，它提供了将整个段分页的访问控制信息。

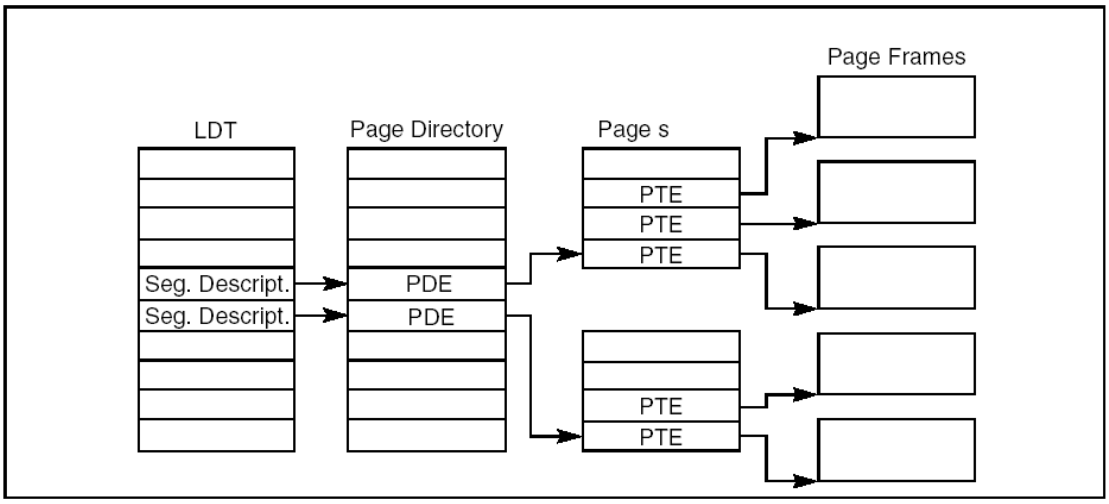

Figure 3-24. Memory Management Convention That Assigns a Page Table to Each Segment

3.11. TRANSLATION LOOKASIDE BUFFERS (TLBS)

处理器将最近使用过的页目录和页表项存放在on－chip的缓存中，这种缓存叫做转换后备缓冲区或者TLBs。P6和奔腾系列处理器中，数据和指令有各自的TLBs。而且在P6系统处理器中，4KB和4MB的页有各自的TLBs。CPUID指令可以用来得知P6和奔腾系列处理器的TLB的大小。

大多数分页操作都是使用TLBs中的内容来进行的。仅当请求页时，进行地址转换所需的信息并不在TLBs中时，才需要为访问内存中的页目录和页表而花费总线周期。

应用程序和一般任务（特权级大于0）是无法访问TLBs的；也就是说，应用程序不能使TLBs无效。只有操作系统和特权级为0的进程可以使TLBs无效或者让选定的TLB无效。每当一个页目录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：

- 显示的，使用MOV指令，比如：

  MOV CR3，EAX

  在这里EAX寄存器包含有一个适当的页目录基址。
- 隐式的，通过执行任务切换来自动改变CR3寄存器的内容。

INVLPG指令是用来使某个在TLB中的页表项失效的。通常，这个指令只是使单个的TLB项失效；然而，有的时候，这条指令能够不仅仅让所指定的项失效，而且能让所有的TLBs失效、

这条指令会忽略页目录或者页表项的G标志（参看下一段落）。

为了避免在任务切换时或者装载CR3寄存器时，TLBs中频繁使用的页被自动置为失效，可以使用（在奔腾pro处理器中引入）CR4寄存器中的“启用全局页”标志和页目录项或页表项中的“全局（G）”标志（位8）。（更多关于“全局”标志的信息，请参看3.7.6节 “页目录项和页表项”）

当处理器为了一个全局页，装载一个页目录或者页表项到TLBs，这个表项将不确定的保留在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，TLB就失效了
- 执行INVLPG指令来使某个TLB中特定的页目录或页表项失效。

更多关于使TLBs失效的信息，请参看10.9.节 “使TLB失效”。

第 4 章 保护模式

在保护模式下，IA32 架构为段级和页级上的操作，都提供了一种保护机制。根据权限，这种保护机制限制了对特定段或者页的访问（段的特权级有 4 级，页有 2 级）。比如，可以将关键性的操作系统代码和数据放在高特权级的段上来保护它们。处理器的保护机制可以防止应用程序的代码不加控制地访问操作系统的代码和数据（The processor's protection mechanism will then prevent application code from accessing the operating-system code and data in any but a controlled, defined manner.）

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。

当使用保护模式时，对内存的任何引用都要进行检验，以确定是否符合权限的要求。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。因为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：

- 限长的检验（limit checks）
- 类型检验
- 特权级检验
- 可寻址范围的限制
- 例程入口点（procedure entry-point）的限制
- 指令集的限制

所有的保护违例都会产生一个异常。关于异常机制的解释，请参看第五章 *中断和异常处理* 。本章主要描述保护模式机制和导致异常的违例。

> 以下几节描述了在保护模式下可用的保护机制。有关实地址下的保护和虚拟 8086 模式的更多信息，请参看 第十六章。

4.1 启用/禁用段和页的保护

把 CR0 寄存器中的 PE 标志位置位，将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，可以基于特权级，将段的保护机制从本质上禁用；具体做法是这样的，将所有段选择符和段描述符中的特权级设为 0（最高特权级）。这个办法可以消除段之间的特权级屏障，但是其他的保护检验，如限长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，页级的保护也就自动启动了。与段保护一样，一旦页保护启用了，就没有控制位可以将页级的保护关闭。然而，通过如下操作可以禁用页级的保护：

- 将 CR0 寄存器中的 WP 标志清零
- 将每个页目录项和页表项的读/写标志和普通用户/超级用户（U/S）标志都置位。

这个行为将每个页都变成了可写的，普通用户级的。这实际上就是使页级的保护失去意义了。

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的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 bit12） 用来确定该段描述符是一个系统段描述符、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 bit8 到 bit11）确定代码段，数据段或者系统段的类型。
- 限长域——（段描述符的第一个双字的 bit0 到 bit15，第二个双字的 bit16 到 bit19）这个域和 G 标志，还有 E 标志（如果是数据段）一起来确定该段的长度。
- G 标志——（段描述符第二个双字的 bit23）它与限长与和 E 标志（如果是数据段）一起来决定该段的长度。
- E 标志——（数据段描述符第二个双字的 bit10），它与限长域和 G 标志一起来决定段的长度。
- 描述符特权级（DPL）域——（段描述符的第二个双字的 bit13 和 bit14）确定该段的特权级。
- 请求特权级（RPL）域——（段选择符的 bit0 和 bit1）确定一个段选择符的请求特权级。
- 当前特权级（CPL）域——（CS 段寄存器的 bit0 和 bit1）指明当前运行的进程的特权级。术语 *当前特权级（CPL）*就是指该域的设置。
- 普通用户/超级用户（U/S）标志——（页目录项或页表项的 bit2）确定该页的类型：普通用户级还是超级用户级。
- 读/写（R/W）标志——（页目录项和页表项的 bit1）确定该页访问类型：只读还是可读可写。

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

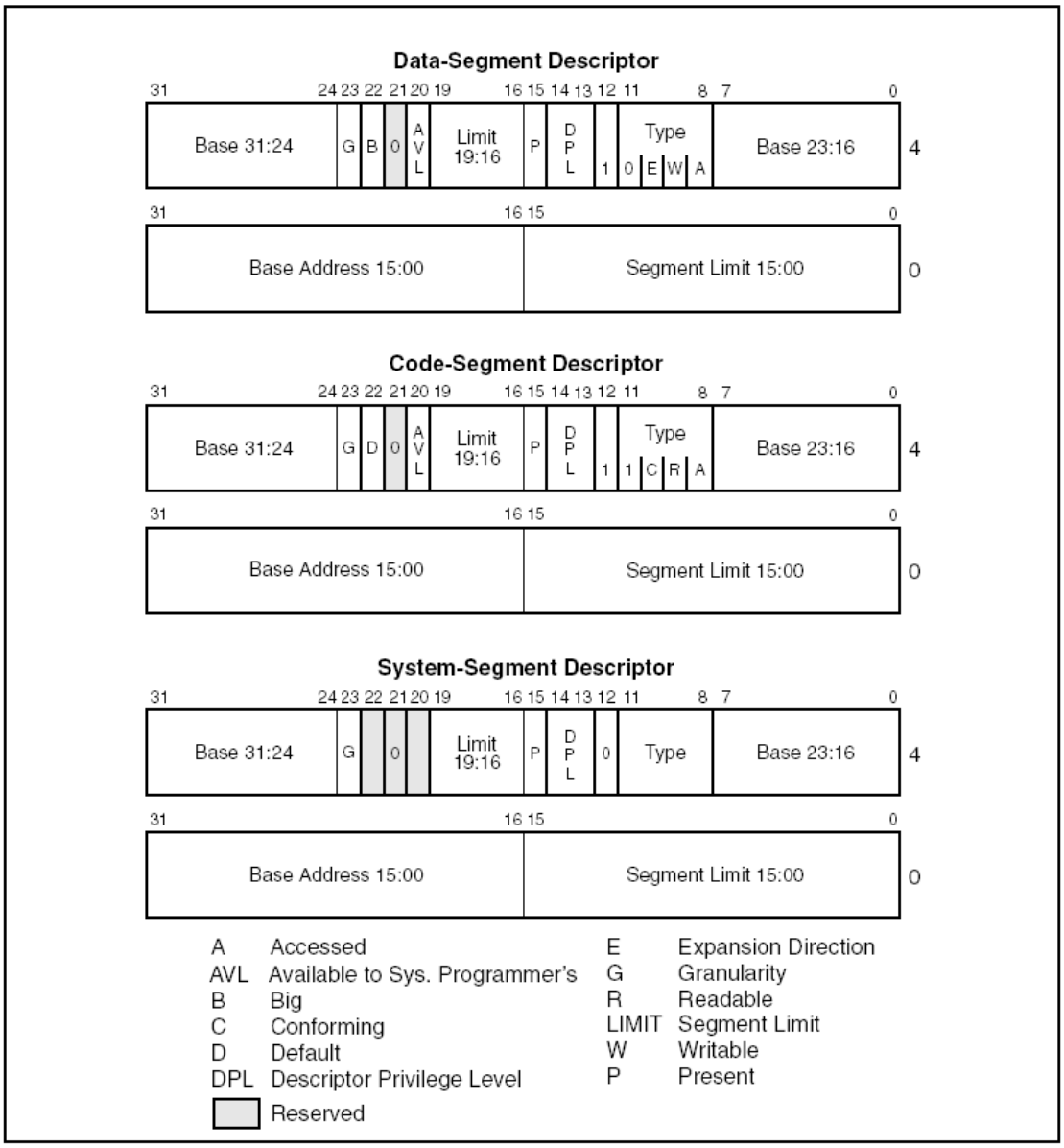

Figure 4-1. Descriptor Fields Used for Protection

通过这些域和标志，可以实现很多不同风格的保护方案。When the operating system creates a descriptor, it places values in these fields and flags in keeping with the particular protection style chosen for an operating system or executive. 应用程序一般不会访问和改变这些域和标志。

下一节描述了处理器如何使用这些域和标志来完成在本章 introduction 中描述的各种检验。

4.3.限长检验

段描述符中的限长域可以防止进程访问段之外的内存。限长的有效值取决于粒度（granularity）G 标志的设置（见图 4－1）。对于数据段，限长还要依赖于扩展方向 expansion direction）E 标志和 B（default stack pointer size and/or upper bound)标志。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byte）时，该段的有效限长就是段描述符中 20bit限长域中的值。此处，限长的值域为 0 到FFFFFH（1MB）。当G标志为 1 时（4KB页粒度），处理器将限长域的值乘上 2^{12}（4KB）作为

段限长的有效值。这种情况下，有效限长的值域为FFFH（4KB）到FFFFFFFFH（4GB）。当G标志为 1 时，就不需要对段偏移量值的低 12bit进行限长的检验了；注意，如果段限长为 0，偏移量从 0 到FFFH仍然是有效的。

除了向下扩展的数据段，有效限长就是该段中被允许访问的最后的地址，它比段的尺寸小 1byte。任何时候，试图访问段中的如下地址时，处理器将产生一个一般保护异常（GP）：

- Byte 偏移量的值大于有效限长
- 字偏移量的值大于有效限长减 1
- 双字偏移量大于有效限长减 3
- 四字偏移量大于有效限长减 7

对于向下扩展的数据段而言，段限长有同样的功能，但是意义却不一样。此处，有效限长定义为段中最后一个不允许访问的地址；如果 B 标志为 1，合法偏移量的范围为（有效限长减 1）到 FFFFFFFFH；如果 B 标志为 0，合法偏移量的范围为（有效限长减 1）到 FFFFH。

限长的检验可以捕捉到某些程序的错误，比如失控的代码和脚本，以及非法指针计算。这些错误是可以在发生的时候被检测到的，所以对错误原因的确认就会更加容易。没有限长的检验，这些程序错误可能会导致另外一个段的代码和数据被覆盖。

除了检验段限长外，处理器还要检验描述符表的限长。GDTR 和 IDTR 寄存器含有 16bit 的限长值，这可以防止进程从描述符表外的地方读取段描述符。LDTR 和任务寄存器含有 32bit 的段限长值（从当前的 LDT 和 TSS 的段描述符中获取的）。处理器使用这些段限长来阻止对超越当前 LDT 和 TSS 界限的访问。更多关于 GDT 和 LDT 限长域的信息，请参看 3.5.1 节 “段描述符表”；更多关于 LDT 限长域的信息，请参看 5.10 节 “中断描述符表”；更多关于 TSS 段限长域的信息，请参看 6.2.3 节 “任务寄存器”。

4.4 类型检验

在段描述符中，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，这些错误可能导致以不恰当的方式使用一个段或者门。

S 标志指明描述符的是系统类型还是代码或者数据类型。类型域提供了额外的 4 个 bit，用来定义各种类型的代码，数据和系统描述符。表 3－1 显示了对代码和数据描述符的类型域的解码；表 3－2 显示了对系统描述符的类型域译码。

每当对段选择符和段描述符进行操作时，处理器就检验类型信息。当要进行如下操作时，需要进行类型检验，以下所列出的操作集合并不完整：

- **当把一个段选择符装载入段寄存器时**。特定的段寄存器只能容纳特定的描述符类型。比如：
  － CS 寄存器只能装入代码段的选择符。
  －不可读的代码段或者系统段的段选择符不能载入数据段寄存器（DS，ES，FS 和 GS）。
  －只有可写的数据段的段选择符才能装载入 SS 寄存器。
- **当段选择符装载入 LDTR 或任务寄存器时**。

－LDTR 只能装 LDT 的选择符
－任务寄存器只能装 TSS 的段选择符。

- **当指令试图访问其描述符已经被装载入段寄存器的段时**。特定的段只能以某些预定的方式访问，比如：
  －没有任何指令可以对一个可执行的段执行写操作
  －没有指令可以对一个不可写的数据段执行写操作
  －除非可读标志被置位，否则没有指令可以读取一个可执行段。
- 当一个指令的操作数含有段选择符时。某些指令只能访问某个特定类型的段或门，比如：
  －远调用或远跳转指令只可以访问这些代码段的描述符，它们是一致代码段，非一致代码段，调用门，任务门或者 TSS。
  －LLDT 指令中只能涉及（must reference）TSS 的段描述符。
  －LAR 指令中只能涉及（must reference）LDT，TSS，调用门，任务门，代码段或数据段的描述符。
  －LSL 指令中只能涉及（must reference）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。
- **某些内部操作期间**。比如：
  ——在一个远调用或一个远跳转（执行远调用或远跳转指令）期间，CALL 或 JMP 指令中的操作数作为段选择符而指向一个段（或者门）描述符；处理器通过检验这个描述符的的类型域来确定即将执行的控制转换(调用或者跳转到另外一个代码段,通过门的调用或者跳转,或者一个任务切换)。如果描述符类型是一个代码段或调用门，表明是调用或者跳转到另外一个代码段；如果描述符类型是一个 TSS 或者任务门，表明进行了一个任务切换。

  ——通过调用门进行的调用或者跳转（或者在一个中断，或者通过陷阱或中断门调用的异常处理程序）时，处理器自动对这个门所指向的段描述符进行检验，以确定该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时（或者在一个中断，或者通过任务门调用异常处理程序时），处理器自动检验该任务门所指向的段描述符是否为一个 TSS。

  ——通过直接指向一个 TSS 进行调用或者跳转到一个新任务时，处理器知道检验 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．对空段选择符的检验

试图将一个空段选择符（见 3.4.1.节 *“段选择符”*）装载入 CS 或 SS 段寄存器会产生一个通用保护异常（GP#）。可以将空段选择符装载入 DS，ES，FS 或 GS 寄存器，但是任何试图通过这些值为空段选择符的寄存器来访问某个段，都将导致一个通用保护（GP）异常的产生。将空段选择符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可以识别 4 个特权级，从 0－3。权值越大，特权越少。Figure 4-2 显示，处理器如

何以保护环的方式来解释这些特权级。

环的中心（为具有最高特权级的代码，数据和栈所保留）通常为关键性软件所有，比如操作系统内核。外部的环给不是很关键的软件用。（仅使用 2 个特权级的系统应该使用特权 0 和特权 3）

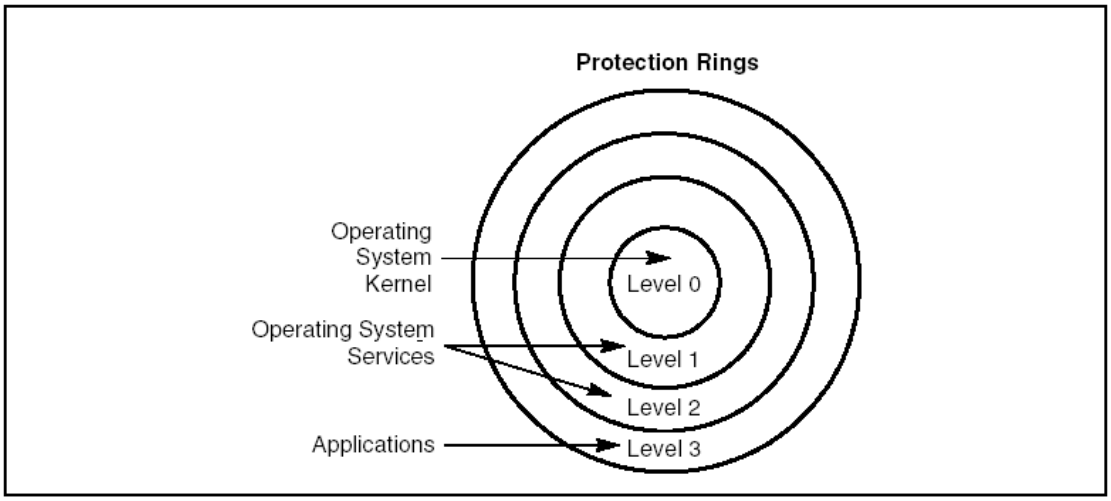

Figure 4-2. Protection Rings

除了某些可控制的情况外，处理器使用特权级来防止较低特权级的进程或任务访问具有较高特权级的段。当处理器检测到一个特权违例，就会产生一个通用保护异常（GP）。

检验代码段和数据段的特权级时，处理器需要识别以下 3 种类型的特权级：

- 当前特权级（CPL）。CPL 是当前正在运行的进程或任务的特权级。它存在 CS 和 SS 段寄存器的 bit0 和 bit1。一般地，CPL 与当前指令所在的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，处理器就会改变 CPL。当访问一致代码段时，对 CPL 的处理会略有不同。当处理器访问一个与 CPL 不一样的特权级的一致代码段时，它不会改变 CPL。
- 描述符特权级（DPL）。DPL 是一个段或门的特权级。它存储在一个段或门的描述符的 DPL 域中。当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符的 RPL 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 指明了可以访问该段的进程或任务特权级。比如，如果数据段的所需要具备的特权级。DPL 是 1 时，只有 CPL 为 0 或 1 的进程才可以访问该段。
  - **非一致代码段（不使用调用门）**。DPL 指明了只能是某个特权级的进程或任务才能访问该段。比如，非一致代码段的 DPL 为 0，那么只有 CPL 为 0 的进程才能访问该段。
  - **调用门** The DPL indicates the numerically highest privilege level that the currently executing program or task can be at and still be able to access the call gate. (This is the same access rule as for a data segment.)
  - **通过调用门访问的一致代码段和非一致代码**　　DPL 指明了能访问该段的进程或任务的 the numerically lowest privilege level。比如，一个一致代码段的 DPL 为 2，那么 CPL 为 0 或 1 的进程就不能访问该段。
  - **TSS** The DPL indicates the numerically highest privilege level that the currently executing program or task can be at and still be able to access the TSS. (This is the same access rule as for a data segment.)
- 请求特权级（RPL） RPL is an override privilege level that is assigned to segment selectors.它存储在段选择符的 bit0 和 bit1。处理器会检验 RPL 和 CPL 来决定对这个段的访问是否是被允许的。即使访

问该段的进程或任务有足够的特权级，如果 RPL 不够，访问也会被拒绝的。也就是说，如果段选择符的 RPL 在数值上比 CPL 大，RPL 就 override CPL，反之亦然。The RPL can be used to insure that privileged code does not access a segment on behalf of an application program unless the program itself has access privileges for that segment. 更详细的关于 RPL 的用途和典型用法，请参看 4.10.4 节，“检验 调用者访问权限（ARPL 指令）”。

当一个段选择符被装载入段寄存器时，将进行特权级的检验。对数据访问的检验与对代码段的进程控制转换的检验是不同的；以下几节中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符装载入数据段寄存器（DS，ES，FS 或 GS）或者装载入堆栈段寄存器（SS）。(可以用如下指令装载段寄存器，MOV，POP，LDS，LES，LFS，LGS 和 LSS 指令)。处理器将段选择符装载入段寄存器之前，它要进行特权级检验（见图 4－3），比较当前运行的进程或任务的特权级（CPL），段选择符的 RPL，还有该段的段描述符的 DPL。如果 DPL 在数值上比 CPL 和 RPL 都大或者相等，处理器会将段选择符装载入段寄存器。否则，处理器会产生一个通用保护错，并且不会装载段寄存器。

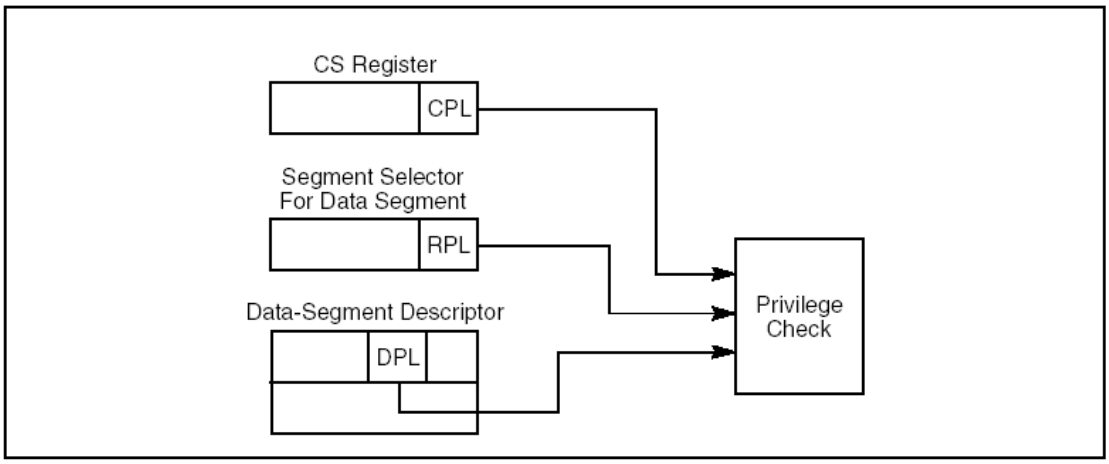

Figure 4-3. Privilege Check for Data Access

图 4－4 显示了 4 个进程（分别在代码段 A，B，C，D），每个进程都运行在不同的特权级，每个进程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符 E1 来访问数据段 E，因为代码段 A 的 CPL 和段选择符 E1 的 RPL 与数据段 E 的 DPL 相等。
- 代码段 B 上的进程可以通过段选择符 E2 来访问数据段 E，因为代码段 B 的 CPL 和段选择符 E2 的 RPL 都在数值上比数据段 E 的 DPL 小（也就是 CPL 和 RPL 具有更高的特权级）。B 代码段的进程也可以通过段选择符 E1 来访问数据段 E。
- 代码段 C 上的进程不能通过段选择符 E3(dotted line)，因为代码段 C 的 CPL 和段选择符 E3 的 RPL 在数值上都比数据段 E 的 DPL 大（意味着较小的特权级）。即使代码段 C 的进程使用段选择符 E1 或 E2，RPL 的够级别了，但是 CPL 的级别不够，仍然不能访问数据段 E。
- 代码段 D 上的进程本应当可以访问数据段 E，因为代码段 D 的 CPL 在数值上比数据段 E 的 DPL 在数值上更小。然而段选择符 E3 的 RPL 在数值上比数据段 E 的 DPL 大，因此该进程对数据段 E 的访问被禁止了。如果代码段 D 上的进程使用段选择符 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。

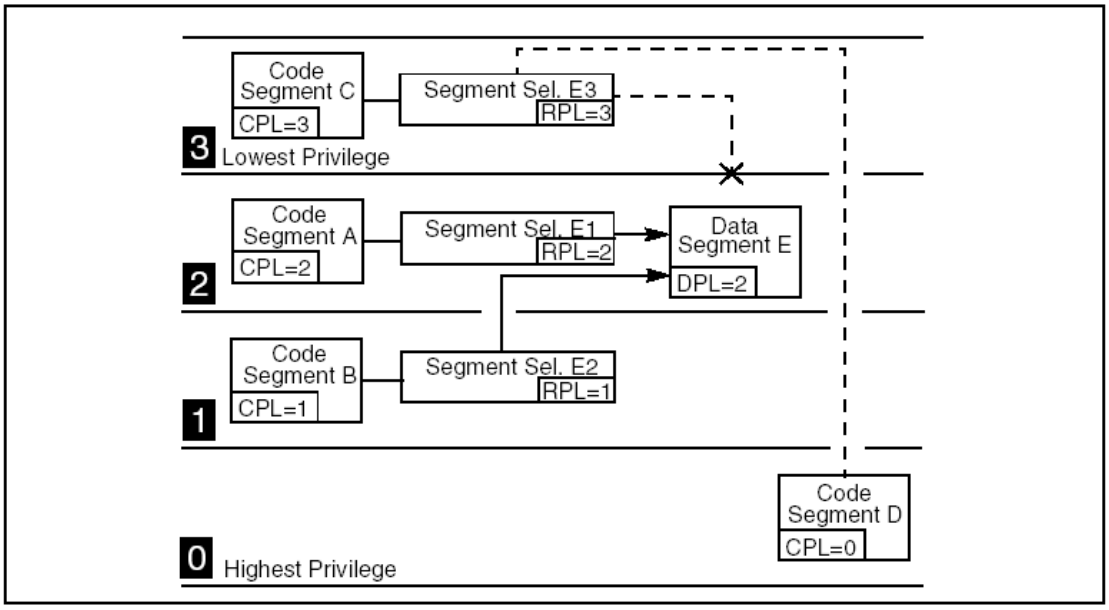

Figure 4-4. Examples of Accessing Data Segments From Various Privilege Levels

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，那么任何数据段都是可访问的；当 CPL 为 1，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。

The RPL of a segment selector can always override the addressable domain of a program or task. 如果使用得当，RPL 可以防止数据段的段选择符被特权级不够的进程或任务有意无意的错误使用。

值得注意的是，数据段的段选择符的RPL是软件控制的。比如，一个CPL为3的应用进程可以将数据段的段选择符的RPL置为0。With the RPL set to 0, only the CPL checks, not the RPL checks, will provide protection against deliberate, direct attempts to violate privilege-level security for the data segment。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符，都要进行访问特权级的检验。

It is important to note that the RPL of a segment selector for a data segment is under software control. For example, an application program running at a CPL of 3 can set the RPL for a datasegment selector to 0. With the RPL set to 0, only the CPL checks, not the RPL checks, will provide protection against deliberate, direct attempts to violate privilege-level security for the data segment. To prevent these types of privilege-level-check violations, a program or procedure can check access privileges whenever it receives a data-segment selector from another procedure (see Section 4.10.4., “Checking Caller Access Privileges (ARPL Instruction)”).

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。可以用以下的的方法访问代码段中的数据：

- 将一个非一致，可读代码段的段选择符装载入数据段寄存器
- 将一个一致，可读代码段的段选择符载入数据段寄存器。
- Use a code-segment override prefix (CS) to read a readable, code segment whose selector is already loaded in the CS register.

访问数据段的 rules 同样适用于方法 1。方法 2 总是正确的，因为无论 DPL 是多少，一致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 寄存器时进行特权级检验

当某个堆栈段的段选择符被装载入 SS 寄存器时，也会进行特权级检验。此处，所有与该堆栈段相关的特权级必须与 CPL 匹配；也就是说，CPL，堆栈段的段选择符的 RPL，它的段描述符的 DPL 必须相等。如果 RPL 和 DPL 与 CPL 不相等，就会产生一个通用保护异常（GP）。

4.8.进程在代码段之间进行控制转换时的特权级检验

当进程的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符装载至 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，对段限长，类型及特权级进行检验。一旦通过检验，CS 寄存器装载完毕，进程的控制就转换到新的代码段，进程的执行就从 EIP 所指向的指令开始。

进程控制转换通常由于 JMP，CALL，RET，SYEENTER，SYSEXIT，INT n，和 IRET 指令或者因异常和中断的产生而发生。异常，中断和 IRET 指令作为特例将在第 5 章 *中断和异常处理* 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符。
- 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符。
- 目标操作数指向一个 TSS，该 TSS 保护目标代码段的段选择符。
- 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符。

下面几节描述了前两种引用方式。通过任务门或 TSS 进行进程控制转换的信息，参看 6.3 节，“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 节 “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 指令的近形式（near form）只是在当前代码段内进行进程控制的转换，所以不进行特权级的检验。而这些指令的远形式（far form）将控制转换到另外一个代码段，所以处理器要进行特权级检验。

当不通过调用门而转换进程控制到另外一个代码段时，处理器检查以下 4 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）

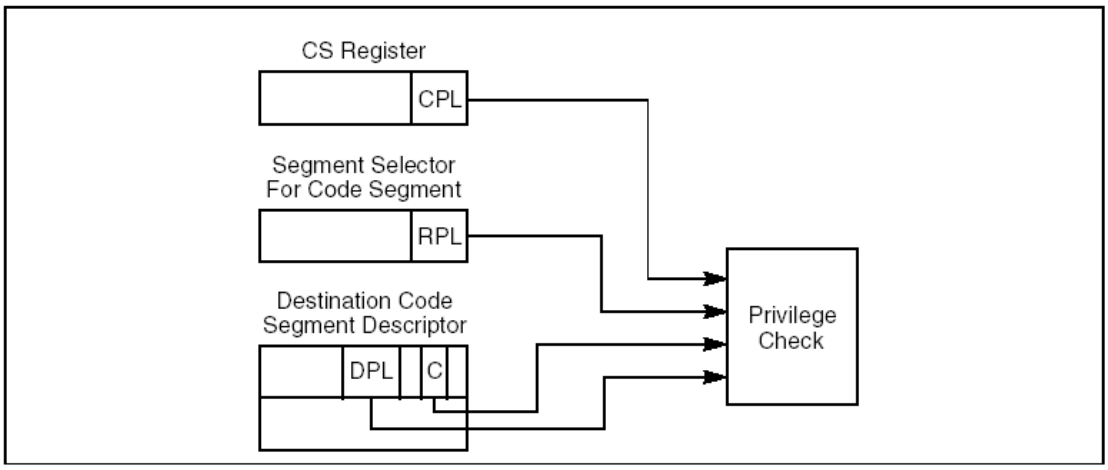

Figure 4-5. Privilege Check for Control Transfer Without Using a Gate

- 被调用例程所在的目标代码段描述符的 DPL。
- 目标代码段段选择符的 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代码段（标志为 1）还是非一致代码段（标志为 0）。（更多关于这个标志的信息，参看 3.4.3.1.节 “代码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. 访问非一致代码段

当访问非一致代码段时，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通用保护异常（GP）。

比如，在图 4－6 中，代码段 C 是一个非一致代码段。那么，代码段 A 中的例程可以调用代码段 C 中的例程（使用段选择符 C1），因为他们有相同的特权级（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（使用段选择符 C2 或 C1）就不能调用代码段 C 中的例程，因为着两个代码段在不同的特权级上。

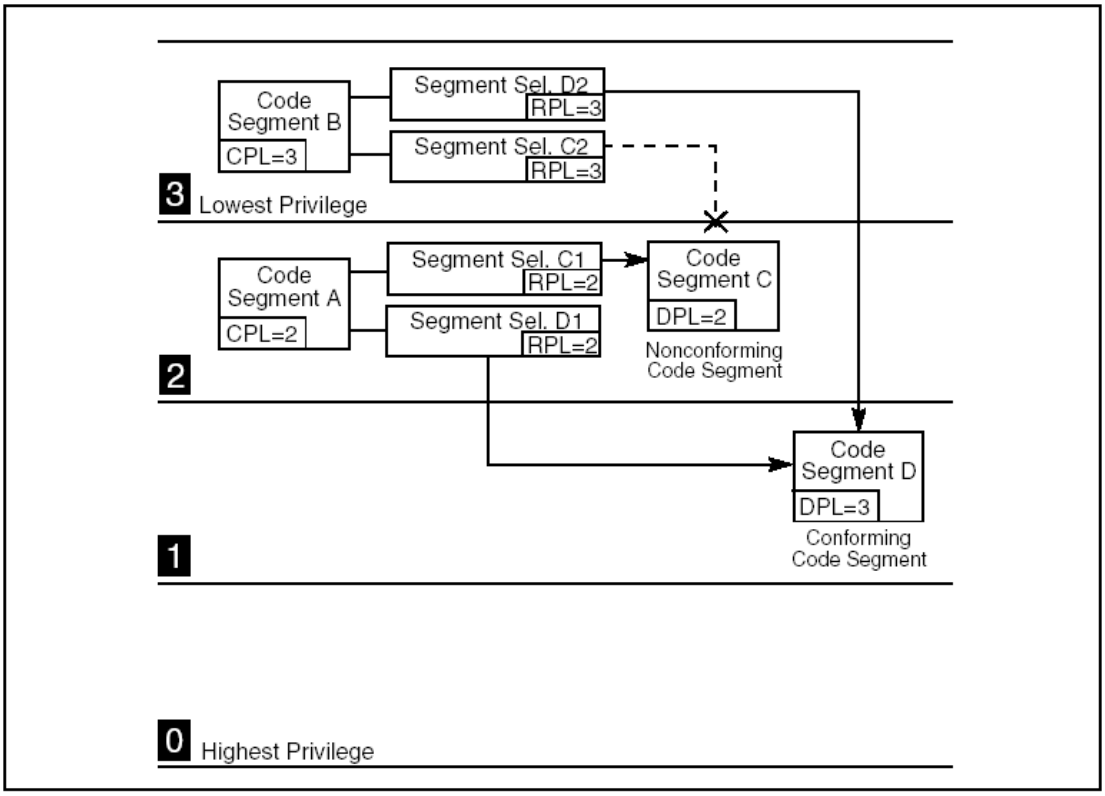

Figure 4-6. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

指向非一致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，以便成功的进行控制转换。所以，在图 4－6 的例子中，段选择符 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 寄存器中装载入一个非一致代码段的段选择符时，不改变特权级域的值；这意味着，调用例程的 CPL 被保留。即使该段选择符的 RPL 与 CPL 不一样时，也是这样。

4.8.1.2. 访问一致代码段

当访问一致代码段时，调用例程的 CPL 可以在数值上与目标代码段的 DPL 相等或大于 DPL（也就是特权比目标代码段的特权低）；仅当 CPL 比 DPL 小的时候，处理器会产生一个通用保护异常（GP）。（当目标代码段是一致代码段时，不用检验目标代码段的 RPL）

在图 4－6 中的例子中，代码段 D 是一个一致代码段。因此，代码段 A 和 B 中的调用例程都可以访问代码段 D（分别 0 使用段选择符 D1 或 D2），因为他们的 CPL 都比这个一致代码段的 DPL 大或者与它相等。（？？？？与图 4－6 不符？？？）对一致代码段而言，当调用例程能成功调用该代码段时，DPL 就是该例程可以拥有的最高特权级。

（注意，段选择符 D1 和 D2 除了 RPL 不一样外，其余的都是一样的。但是当访问一致代码段时，并不检验 RPL，因此，在这种情况下，这两个段选择符本质上是一样的）

当进程控制转入一个一致代码段时，即使目标代码段的 DPL 级别高于 CPL，也不改变 CPL（译者注：CS 寄存器中的特权级域不变，即装载新代码段的选择符到 CS 中时，CPL 域保持不变）。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的一部分，但是他们可以以更低特权级来执行。当进程切换到一个一致代码段时，保留当前进程的 CPL 可以防止应用程序在一致代码段的特权级上访问非一致代码段，同时防止它访问更高特权级的数据。

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

4.8.2.门描述符

为了能够访问不同特权级的代码段，处理器提供了一个特殊的描述符集合，叫做门描述符。以下是四种门描述符：

- 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。它们在第五章 *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换提供了便利。它们一般只用在操作系统中或者使用特权级保护机制的程序里。调用门也可用于 16bit 和 32bit 代码段之间的进程控制转换，就如 17.4 节 “混合尺寸代码段之间的控制转换” 中所描述的。

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（IDT）中。它完成六种功能：

- 确定将要访问的代码段
- 定义在指定代码段中的一个例程入口
- 指明了调用该例程的所应当有的特权级

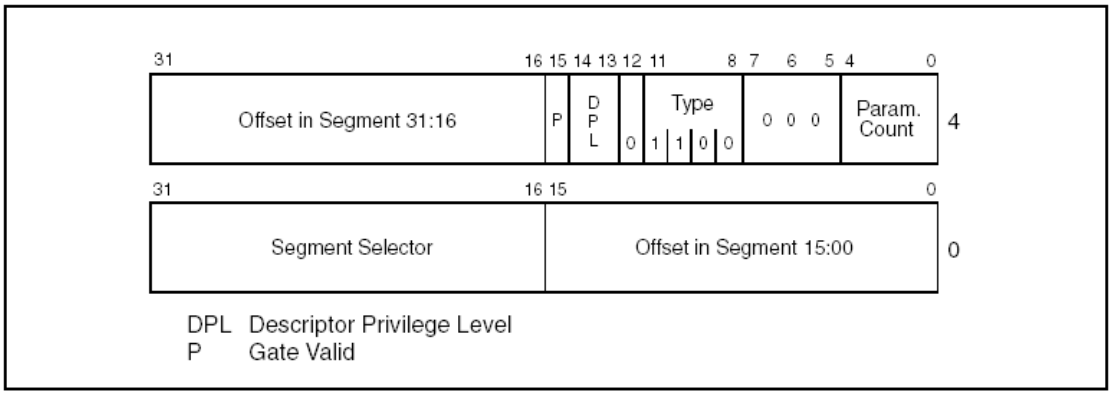

Figure 4-7. Call-Gate Descriptor

- 如果有栈切换，要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：16bit 的门执行 16bit 的压栈，32bit 的门执行 32bit 的压栈。
- 确定该调用门描述符是否有效

调用门中的段选择符域确定了将要访问的代码段。偏移量域确定了在该代码段中的入口点。这个入口点通常是某个例程的第一条指令。DPL 域指明了该调用门的特权级，也就是通过该调用门访问该例程所必须具备的特权级。P 标志指明该调用门描述符是否有效。（该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，当发生了栈切换时，需要从调用进程的栈拷贝到新栈的参数个数（见 4.8.5.节 “栈切换”）。对于 16bit 调用门来说，参数计数域确定了需要拷贝的 word 数，对于 32bit 调用门来说，就是需要拷贝的 doubleword 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，当一个进程试图访问该描述符，将会产生一个 不存在 （not present ＃NP）异常。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。这样，P 标志通常最初被置为 0 以陷入 not present 异常处理程序。这个异常处理程序就将一个计数器加 1 并且将 P 标志置 1，所以，从这个异常处理程序返回后，这个门描述符就是有效的了。

4.8.4.通过调用门访问代码段

为了访问一个调用门，CALL 或 JMP 指令的目标操作数中有一个远指针指向这个门。这个指针里的段选择符可以识别这个调用门（见图 4－8）；这个指针里还要有偏移量，只是处理器并不检查。（这个偏移量值可以被置为任意值）

一旦处理器访问了这个调用门，它使用这个调用门中的段选择符来定位目标代码段的段描述符。（这个段描述符可以在 GDT 中或者 LDT 中。）之后，结合描述符中的段基址和调用门中的偏移量就可以组成一个线性地址，这就是这个例程在代码段中的入口点的线性地址。

如图 4－9 中所示，一个通过调用门进行的进程控制转换的有效性需要检验四个不同的特权级：

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

同时还要检验一下目标代码段的段描述符中的一致性标志（C flag）标志。

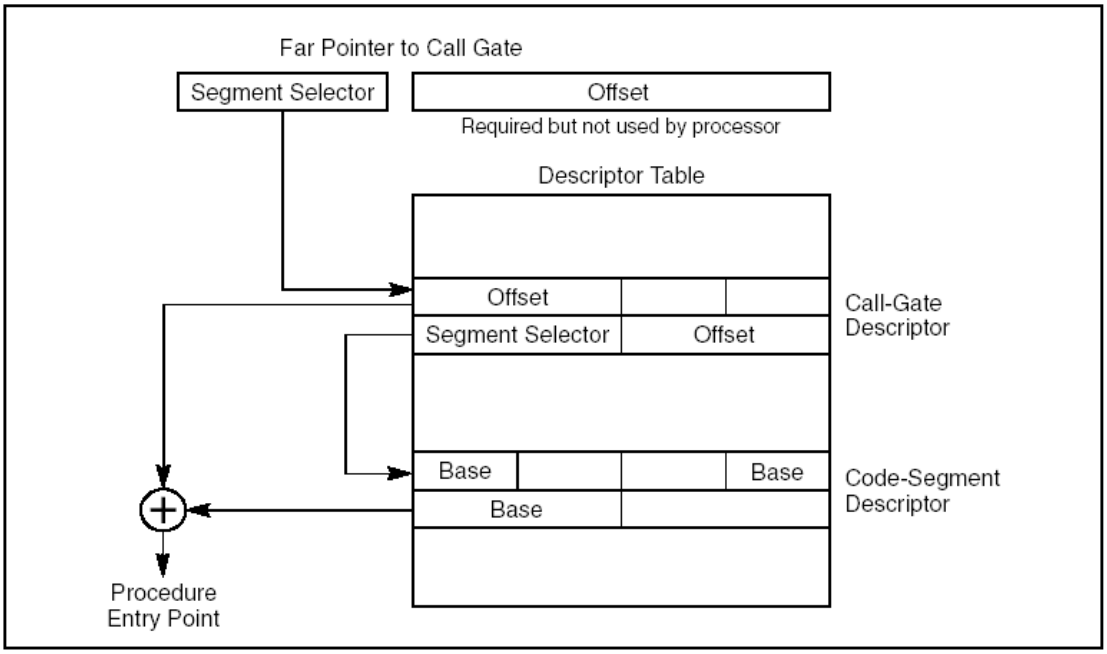

Figure 4-8. Call-Gate Mechanism

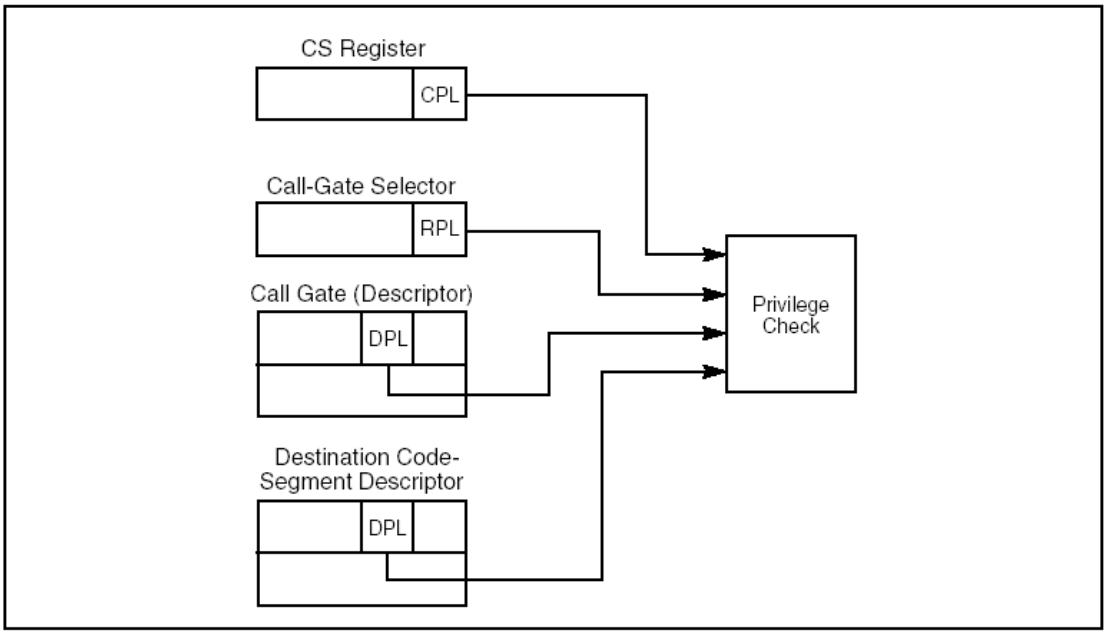

Figure 4-9. Privilege Check for Control Transfer with Call Gate

如表 4－1 所示，特权级检验的准则依赖于进程控制转换的指令是 CALL 还是 JMP 指令。

Table 4-1. Privilege Check Rules for Call Gates

Instruction	Privilege Check Rules
CALL	CPL ≤ call gate DPL; RPL ≤ call gate DPL Destination conforming code segment DPL ≤ CPL Destination nonconforming code segment DPL ≤ CPL
JMP	CPL ≤ call gate DPL; RPL ≤ call gate DPL Destination conforming code segment DPL ≤ CPL Destination nonconforming code segment DPL = CPL

调用门描述符的 DPL 域指定了调用进程能够访问调用门应当具有的最小特权级；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。所以任意 CPL（特权级 0－3）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。

调用门段选择符中的 RPL 必须满足同样的条件，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符 B2 或 B1 访问调用门 B，但是不能使用门选择符 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准则是不同的。只有 CALL 指令可以使用调用门将进程控制转换到一个更高特权级的非一致性代码段；也就是说，可以访问一个 DPL 比 CPL 小的非一致代码段。JMP 指令仅能使用调用门将进程控制转换到一个 DPL 与 CPL 相等的非一致代码段。CALL 和 JMP 指令都可以将进程控制转换到一个更高特权级的一致代码段；也就是，转换到一个 DPL 小于或等于 CPL 的一致代码段。

如果调用更高优先级的非一致性目标代码段，CPL 降低为目标代码段的 DPL 特权级，并且会发生栈切换（见 4.8.5 节 “栈切换”）。如果调用或者跳转到一个更高特权级的一致目标代码段，CPL 不会发生变化，也不会发生栈切换。

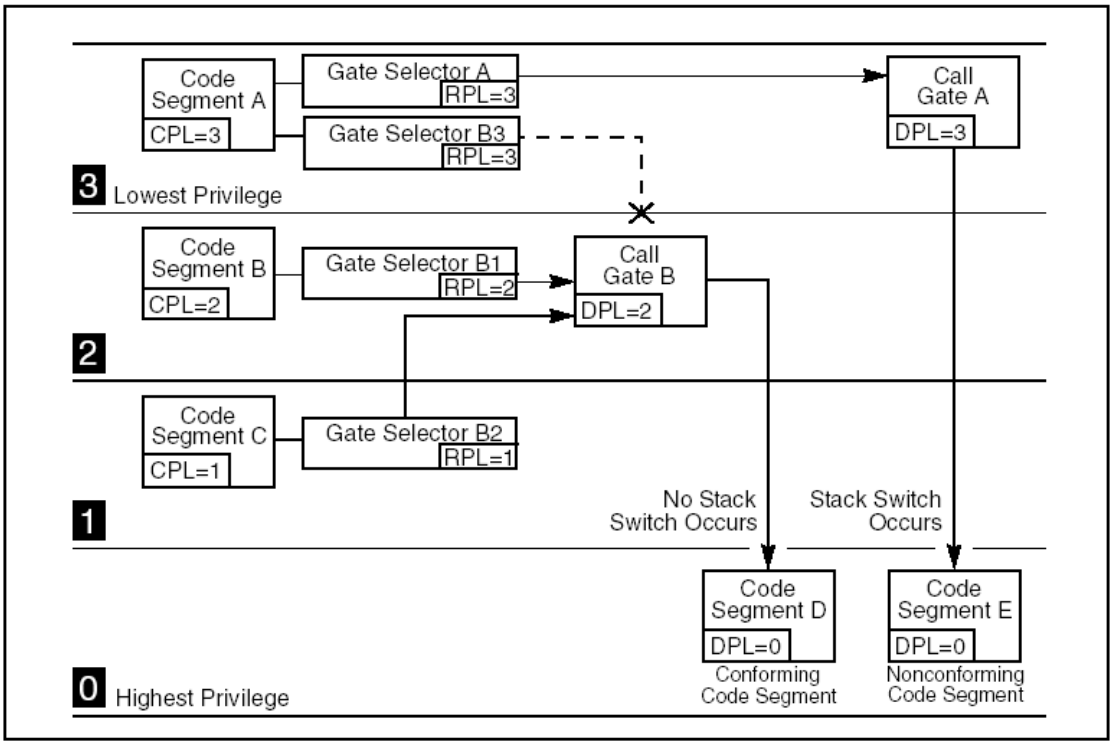

Figure 4-10. Example of Accessing Call Gates At Various Privilege Levels

调用门使得某个代码段上的例程被不同特权级的进程访问。比如，操作系统会装载一个代码段，这个代码段里有一些即为操作系统也为应用软件服务的例程（比如字 I/O 处理例程）。可以为某些例程建立调用门，使得可以在所有的特权级（0－3）访问它们。可以为一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)建立更高特权级的调用门（DPL 为 0 或 1）。

4.8.5.栈切换

一旦调用门被用来转换进程控制到一个更高特权级的非一致代码段（也就是，非一致目的代码段的 DPL 比 CPL 小），the processor automatically switches to the stack for the destination code segment's privilege level。进行这样的栈切换主要是为了防止更高特权级的例程因为栈空间的不足而崩溃。同时这也可以防止较小特权级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 个特权级：3 和 0，就只能定义 2 个栈）每个栈都在一个单独的段中，由一个段选择符和段中的偏移量（栈指针）。

当执行特权级为 3 代码时， 特权级为 3 的栈的栈选择符和栈指针放在 SS 寄存器中和 ESP 寄存器中。The segment selector and stack pointer for the privilege level 3 stack is located in the SS and ESP registers, respectively, when privilege-level-3 code is being executed and is automatically stored on the called procedure's stack when a stack switch occurs.

特权级为 0，1 和 2 的栈的指针存在 TSS 中，为当前运行的任务所用（见图 6－2）。这些指针由一个段选择符和一个栈指针（载入在 ESP 寄存器中）组成。这些初始的指针都是严格的只读的。在这个任务运行时，处理器不会改变它们的值的。它们只是用来在调用更高特权级时创建新的栈。这些栈在从被调用的例程返回时被释放。下次这个例程被调用，可以使用这初始的栈指针创建新的栈。（TSS 并不为特权级 3 的进程指定栈，因为处理器只允许在例程返回时将进程控制权从 CPL 为 0，1，2 的运行例程转换到 CPL 为 3 的运行例程，否则是不允许的）

操作系统要负责创建栈和所有特权级使用的栈段的描述符，还要负责将那些栈的初始指针载入 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（在限长域中定义）来持有如下这些数据：

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

栈需要有足够的空间来包含许多帧上述所列的内容，因为一个例程经常会调用其他的例程，并且操作系统可能支持多重中断的嵌套。每个栈必须在它的特权级内足够大，这样才能顾及最糟糕的情况。

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用的例程：

1. 使用目标代码段的 DPL（也就是新的 CPL）从 TSS 中选中一个指向新栈的指针（段选择符和栈指针）。
2. 从当前 TSS 中读取将要切换到的栈的段选择符和栈指针。在读入栈段选择符，栈指针或栈段描述符时，如果检查到任何限制违例，都会产生一个非法 TSS（＃TS）异常
3. 检查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
4. 暂时保存当前 SS 和 ESP 寄存器的值
5. 将新栈的段选择符和栈指针载入 SS 和 ESP 寄存器
6. 将暂时保存的 SS 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 和 EIP 寄存器）压入新栈

9. 将新代码段的段选择符和调用门的新指令指针分别载入 CS 和 EIP 寄存器，然后开始指向调用例程。有关通过调用门进行远调用时的特权级检验和其他保护检验的相关信息，请参看 第二卷 第三章中关于调用指令的描述。

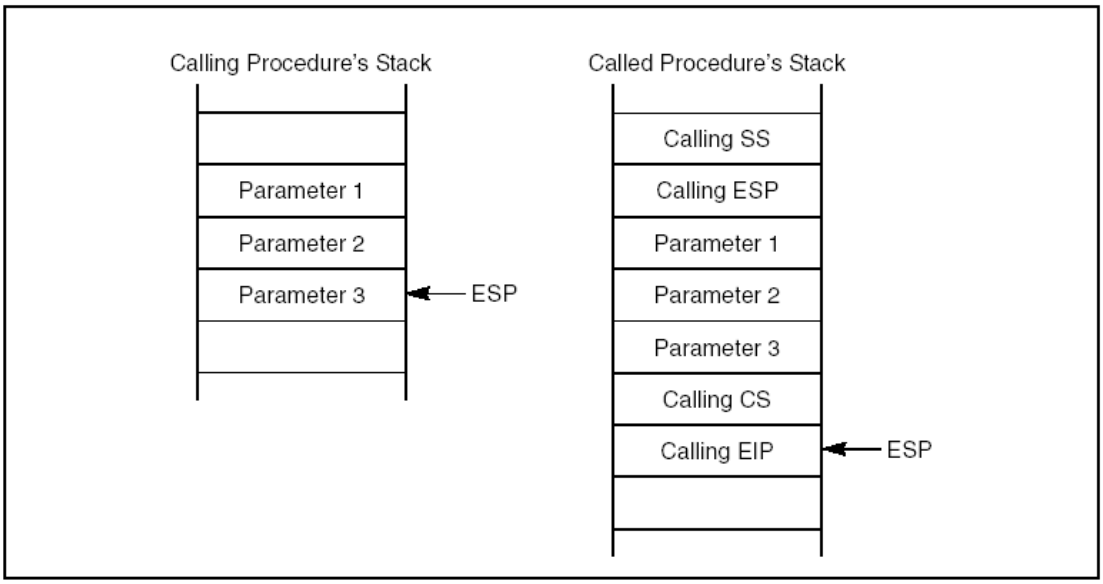

Figure 4-11. Stack Switching During an Interprivilege-Level Call

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用例程的栈的数据个数（最多 31 个）。如果有超过 31 个数据项要传给被调用例程时，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 和 ESP 寄存器的值来访问在老栈空间的参数。如 4.8.3 节 “调用门”中所述，传给被调用例程的数据项的长度取决与调用门的长度。

4.8.6.从被调用例程返回

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。这个指令就是为了执行从一个由 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。

近返回仅仅是在当前代码段内进行控制转移；这种情况下，处理器仅仅做限长的检验。当处理器将返回的指令指针弹入 EIP 寄存器时，它会检验这个指针是否超出了当前代码段的限长。

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符，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，因为他们都是被 CALL 指令压入栈中。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，只能返回到较小的特权级（也就是，返回到的代码段的 DPL 在数值上比 CPL 大）。处理器使用 CS 寄存器中的 RPL 域来确定是否可以返回到数值上较高的特权级。如果 RPL 的特权比 CPL 低，a return across privilege levels occurs.

当执行远返回到调用例程时（关于返回前和返回后栈中的内容，参看 卷 1 的图 6－2 和图 6－4），处理

器进行如下步骤的操作：

1. 检查保存的 CS 寄存器的 RPL 域来确定返回时是否需要特权级的变化
2. 从被调用例程的栈上载入 CS 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符的 RPL 上进行的）
3. （如果 RET 指令包含一个参数计数操作数，并且该返回要求改变特权级）Adds the parameter count (in bytes obtained from the RET instruction) to the current ESP register value (after popping the CS and EIP values), to step past the parameters on the called procedure's stack. The resulting value in the ESP register points to the saved SS and ESP values for the calling procedure's stack. (Note that the byte count in the RET instruction must be chosen to match the parameter count in the call gate that the calling procedure referenced when it made the original call multiplied by the size of the parameters.)
4. （如果该返回要求改变特权级）将保存的 SS 和 ESP 值载入 SS 和 ESP 寄存器，并切回到调用例程的栈。被调用例程的栈的 ESP 和 SS 值就被丢弃了。载入栈段选择符或栈指针时，检测到任何限长违例都会产生一个 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器的操作数）将 ESP 寄存器的值加上这个参数计数器，跳过在调用例程栈上的参数。此时并不检验 ESP 中的值是否超过了该栈段的限长。如果 ESP 的值超过了段限长，直到下一次栈操作才能被发现。
6. （如果该返回需要特权级的变化）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（包括一致代码段），该寄存器就会被赋予一个空的段选择符。

有关在远返回时，处理器进行特权级检验和其他保护检验的更详细描述，请参看卷 2 第三章中关于 RET 指令的描述。

4.8.7.使用 SYSENTER 和 SYSEXIT 指令对系统例程进行快速调用

SYSENTER 和 SYSEXIT 指令是在奔腾 2 处理器时引入 IA32 架构的，目的是提供一种调用操作系统的快速机制（低开销）。SYSENTER 指令就是为了特权级 3 的用户代码访问操作系统的资源。SYSEXIT 指令就是为了特权级为 0 的操作系统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 执行；SYSEXIT 指令只能在特权级 0 执行。

SYSENTER 和 SYSEXIT 指令是结伴的指令，但是他们并不构成一对 调用/返回，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。

The target instruction and stack pointer for these instructions are not specified through instruction operands. Instead, they are specified through parameters entered in several MSRs andgeneral-purpose registers.这些指令的目标指令和栈指针不是通过指令操作数来确定的。相反，他们通过在 MSRs 中和几个通用寄存器中的参数来确定。对于 SYSENTER 指令而言，处理器通过以下的来源来获取特权级 0 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－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
- 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

- 栈指针－从 ECX 寄存器中读取。

SYSENTER 和 SYSEXIT 指令能够实现快速调用和返回是因为它们强迫处理器进入一个预定好的状态，当执行 SYSENTER 指令时，处理器进入预定的特权级 0 状态，当执行 SYSEXIT 指令时，处理器进入预定的特权级 3 状态。通过强制预定的和一致的处理器状态，大大减少了通常在远调用时，进入另外一个特权级时的特权级检验。同时，通过将目标上下文状态存放在 MSR 和通用寄存器中，只要不获取目标代码，就可以避免了访问内存。

如果还有什么状态需要保存，以便返回到调用例程，就需要显示地由调用例程保存或者编程约定好。

4.9.特权指令

某些系统指令称为特权指令，是因为它们不能被应用程序使用。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，将会产生一个通用保护异常（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－Read Model-Specific Registers.
- WRMSR—Write Model-Specific Registers.
- 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 IA32 处理器家族中才出现的（见 18.10 节，“奔腾及以后的 IA－32 处理器中的新指令”）。寄存器 CR4 中的 PCE 和 TSD 标志（bit4 和 bit2）允许使用 RDPMC 和 RDTSC 指令，这两条指令可以在任何 CPL 下执行。

4.10.指针验证

在保护模式下操作时，处理器会验证所有的指针，强迫段之间的保护，与特权级的检验独立进行。指针验证要进行如下检查：

1. 检查访问权限 Checking access rights to determine if the segment type is compatible with its use.
2. 检查读写权限
3. 检查指针的偏移量是否超出了段限长。
4. 检查指针的使用者是否可以访问该段

5. 检查偏移量的对齐

在指令执行期间，处理器自动执行第一，二，三步检查。软件必须显示的通过调用 ARPL 指令来请求第四步检查。如果对齐检查在特权级 3 上开启的话，则第五步检查是自动进行的。Offset alignment does not affect isolation of privilege levels.

4.10.1.检查访问权限（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查。这个检查要确定，该段描述符的类型和 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一致或非一致的代码段，调用门，任务门或者一个 TSS。而且，如果调用的是一个非一致代码段，该代码段的 DPL 必须与 CPL 相等，代码段的选择符的 RPL 必须小于或等于 DPL。如果发现类型或者特权级不兼容，就会产生相应的异常。

为了防止类型不兼容异常的产生，程序可以使用 LAR（load access rights）指令来检查某个段的访问权限。LAR 指令需要被检查段描述符的段选择符和一个目的寄存器，它执行如下操作：

1. 检查该段选择符是否为空
2. 检查该段选择符指向的段描述符是否在描述符表（LDT 或 GDT）的限长内
3. 检查这个段描述符是否是一个代码段，数据段，LDT 段，调用门，任务门或者 TSS 段类型
4. 如果该段是一个非一致代码段，就要检查该段描述符是否在当前 CPL 下可见的（也就是，CPL 和段选择符的 RPL 是否小于或等于 DPL）
5. 如果特权级和类型检查都通过了，就将段描述符的第二个双字载入到目的寄存器（masked by the value 00FXFF00H, where X indicates that the corresponding 4 bits are undefined)并且设置 EFLAGS 寄存器中的 ZF 标志。如果在当前的特权级上，段选择符是不可见的，或者，段选择符对 LAR 指令而言是无效的，这个指令就不会改变目的寄存器，也不会 clear ZF 标志。

一旦装载了目的寄存器，程序就可以进行进一步的关于访问权限的检查。

4.10.2.检查读写权限（VERR 和 VERW 指令）

当处理器访问代码或数据段时，它首先检查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）指令来检查读写权限。这两个指令的操作数都是被检查段的段选择符。它们执行如下操作：

1. 检查段选择符是否为空
2. 检查段选择符指向的段描述符是否在描述符表的限长内（GDT 或 LDT）。
3. 检查这个段描述符是一个代码段还是数据段类型
4. 如果该段是一个非一致代码段，还要检查该描述符在当前的 CPL 下是否可见（也就是 CPL 和段选择符的 RPL 是否小于等于 DPL）。
5. 检查该段是否可读或可写。

如果该段在当前 CPL 下是可见和可读的，VERR 指令就会设置 EFLAGS 寄存器中的 ZF 标志；如果该段在当前 CPL 下是可见和可写的（代码段永远不可写），VERW 指令会设置 ZF 标志。当然，如果这些检查中有失败的，ZF 标志就会被清零。

4.10.3.检查指针的偏移量是否在段限长内（LSL 指令）

当处理器访问一个段时，要进行限长的检查以确定偏移量在段限长内。软件可以使用 LSL（load segment limit）指令来进行限长的检查。与 LAR 指令一样，LSL 指令的操作数需要有段选择符和一个目标寄存器。紧接着指令执行如下操作：

1. 检查段选择符是否为空
2. 检查这个段选择符所指向的段描述符是否在描述符表的限长内（GDT 或 LDT）
3. 检查该段描述符的类型：代码段，数据段，LDT 或 TSS 段类型
4. 如果该段不是一个一致代码段，还要检查该段描述符在当前 CPL 下是否可见（也就是，CPL 和段选择符的 RPL 是否小于 DPL）。
5. 如果特权级和类型检查都通过了，将限长（the limit scaled according to the setting of the G flag in the segment descriptor)载入目标寄存器并置位 EFLAGS 寄存器中的 ZF 标志。如果段选择符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符，指令就不会修改目标寄存器，并将 ZF 标志清零。

一旦限长被装载入目标寄存器，软件就可以将段限长与指针中的偏移量进行比较了。

4.10.4.检查调用者的访问权限（ARPL 指令）

段选择符的 RPL 域是为了能够将调用者的特权级（CPL）带到被调用例程。被调例程使用 RPL 来确定是否可以访问某个段。RPL 将被调用例程的特权级降至 RPL。

操作系统例程经常使用 RPL 来防止较低特权级的应用程序访问在较高特权级段内的数据。当一个操作系统例程（被调用例程）从应用程序（调用例程）收到一个段选择符时，它将调用例程的特权级设置为 RPL。接着，操作系统使用这个段选择符访问与它自己相关的段，处理器使用调用例程的特权级（RPL）进行特权级检查，而不是使用操作系统例程的特权级。The RPL thus insures that the operating system does not access a segment on behalf of an application program unless that program itself has access to the segment.

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）访问一个指向具有更高特权级的数据结构（也就是，这个数据结构位于特权级 0 数据段 D 中）的段选择符。

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）可以访问。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，将一个段选择符 D1 放在栈上作为参数传给操作系统。在传这个段选择符前，应用程序将段选择符中的 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符 D1 来访问数据段 D，处理器会比较 CPL（调用之后是 0），段选择符 D1 的 RPL 以及数据段 D 的 DPL（这里是 0）。因为 RPL 比 DPL 大（值越大，特权级越低），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（也就是段选择符中的 RPL）比数据段 D 的 DPL 大（值越大，特权级越低）。

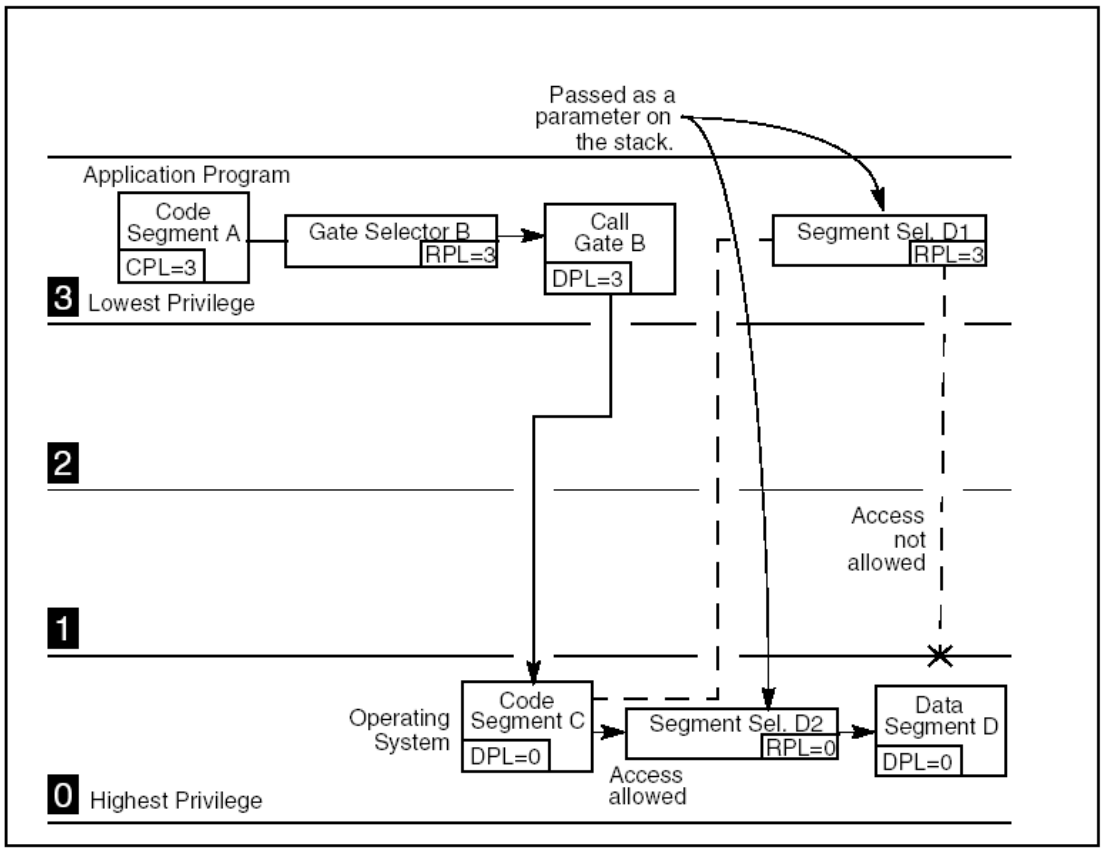

Figure 4-12. Use of RPL to Weaken Privilege Level of Called Procedure

现在假设应用程序不是将段选择符的 RPL 设置为 3，而是将 RPL 设置为 0（段选择符 D2）。现在操作系统可以访问数据段 D，因为它的 CPL 和段选择符 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符的 RPL 设置为任何值，因此它就有可能使用操作系统的例程（其特权级的值比较小，也就是特权级比较高）来访问受保护的数据结构。这种改变段选择符 RPL 的能力破坏了处理器的保护机制。

因为被调用例程不能依赖调用例程来正确的设置 RPL，操作系统例程（运行在较高特权级）从较低特权级的例程收到段选择符以后，需要测试一下段选择符的 RPL 来确定特权级是否合适。ARPL（adjust requested privilege level）指令就是出于这个目的。这个指令会修正段选择符中的 RPL 来匹配另外一个段选择符的 RPL？？？？（This instruction adjusts the RPL of one segment selector to match that of another segment selector.）。

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符 D2 时，它使用 ARPL 指令来比较该段选择符的 RPL 和应用程序的特权级（在栈上的代码段选择符中）。如果 RPL 比应用的特权级数值小，也就是特权级比较高，ARPL 指令就会改变 RPL 来匹配应用程序的特权级（段选择符 D1）。使用这个指令可以防止特权级低的例程通过改小段选择符的 RPL 来访问高特权级的段。

应用程序的特权级可以通过应用程序代码的段选择符的 RPL 域来确定。这个段选择符存放在栈上作为调用操作系统的一部分（This segment selector is stored on the stack as part of the call to the operating system）。操作系统可以从栈上 copy 这个段选择符到一个寄存器中作为 ARPL 的一个操作数。

4.10.5.检查对齐

当 CPL 为 3 时，通过设置 CR0 寄存器中的 AM 标志和 EFLAGS 寄存器中的 AC 标志可以检查内存引用的对齐。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查时的对齐要求。

4.11.页层次的保护

页层次的保护可以单独应用，也可以应用在段上。当在 flat memory model 下使用页层次的保护时，可以将超级用户的代码和数据（操作系统）与普通用户的代码和数据（应用程序）分离开。还可以将有代码的页设置写保护。段层次和页层次的保护机制结合在一起的时候，页层次的读写保护可以使段内的保护粒度更细了。

当使用页层次保护时（同时使用段层次保护），对每个内存引用都要检查，以确保满足保护检查。所有的检查都在内存周期开始之前完成，任何违例都会阻止内存周期启动并导致内存页异常（page fault except）。因为这些检查与地址转换同时进行，不会有任何性能损失。
处理器要进行 2 个页级保护的检查：

- 可寻址范围的限制（超级用户模式和普通用户模式）
- 页类型（只读或可读可写）。

在检查过程中有任何违例都会导致页错误异常。有关页错误异常机制的详细解释，参见第五章，“中断 14－页错误异常（＃PF）”。本章描述了保护违例导致的页错误异常。

4.11.1.页保护标志

页的保护信息包含在页目录项或页表项（见表 3－14）：读/写标志（bit 1）和 user/supervisor 标志（bit2）。第一级和第二级的页表（也就是，页目录和页表）都要需要进行保护检查。

4.11.2.限定可寻址范围

基于 2 个特权级，页层次的保护机制允许对页的访问进行限定：

- 超级用户模式（U/S 标志为 0）－（最高特权级）用于操作系统或其他系统软件（比如驱动程序），以及受保护的系统数据（比如页表）。
- 普通用户模式（U/S 标志为 1）－（最低特权级）用于应用程序代码和数据。

段特权级与页特权级的匹配如下。如果处理器当前运行在 CPL0,1,2, 它处在超级用户模式；如果在 CPL3，它处在普通用户模式。当处理器在超级用户模式时，它可以访问所有的页；当处于普通用户模式时，它只能访问普通用户级的页。（注意，控制寄存器中的 CR0 的 WP 标志会改变 the supervisor permissions, as described in Section 4.11.3., “Page Type”.)

要使用页级保护机制，代码和数据段必须建立至少 2 级基于段的特权级：level0 用于超级用户代码和数据段，level3 用于普通用户的代码和数据段。（这种模型下，栈位于数据段中）。为了减少对段的使用，可以使用 flat 内存模型（见 3.2.1 节，“基本平坦模型”）。

这里，普通用户和超级用户的代码段，数据段都是从线性地址 0 开始的，并且是互相重叠的。这种情况下，操作系统代码（在超级用户级运行）和应用代码（运行在普通用户级）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完成。

4.11.3.页类型

页层次的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）。

当处理器处于超级用户模式并且 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）。当处理器处于普通用户模式，它只能写具有读写访问权限的用户模式页。用户模式页是可读写或者是只读的；超级模式页对用户模式来说是既不可读也不可写的。任何违反这些保护规则的企图都会产生一个页错误异常。

在 P6 系列，奔腾和 intel486 处理器可以禁止用户模式的页被超级用户访问。Setting the WP flag in register CR0 to 1 enables supervisor-mode sensitivity to user-mode, write protected pages.无论 WP 如何设置，超级用户的只读页在任何特权级下都是不可写的。这种超级用户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，spawning）。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每个任务创建一个这个页集合的私有复制。使用 WP 标志并标志这些共享的页为只读，超级用户可以检测到对普通用户页写操作的企图，因此能够在这个时候及时复制页。

4.11.4.两个层次页表的保护相结合

对于任何一个页，它的页目录项（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项和页表项都进行保护检查。表 4－2 显示了当 WP 标志被清零后，所有保护属性的组合所提供的保护。

4.11.5.overrides to 页保护

无论当前处理器处于什么 CPL，以下类型的内存访问需要进行检查，就好像它们在特权级 0 访问似的：

- 对 GDT，LDT 或 IDT 中的描述符的访问
- Access to an inner-privilege-level stack during an inter-privilege-level call or a call to in exception or interrupt handler, when a change of privilege level occurs.

4.12 页保护和段保护相结合

当开启分页时，处理器首先 evaluate 段保护，然后再 evaluate 页保护。如果处理器在段层次或者页层次检测到保护违例，就不能访问内存并且产生一个异常。如果在段层次产生异常，就不会产生分页异常。

页层次的保护不能超越段层次的保护。比如，一个代码段被定义为不可写的。如果该代码段被分页，并

且把这些页设置为可读写的，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层次的保护检测时被阻止。

页层次的保护可以用来强化段层次的保护。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。

Table 4-2. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read-Only
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read-Only
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read-Only
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

注意：

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

第 5 章　中断和异常处理

(这一章是 wykr3879 翻译的,感谢他的辛苦工作!)

声明:
红字体是感觉翻的不好的，或是不确切是否该如此翻译。其后是原文。
直接使用原文则是不知该如何翻☹

这一章描述在保护模式下的处理器处理中断和异常的机制。这里提到的绝大多数内容同样适用于实地址(real-address)和虚拟-8086 模式(virtual-8086 mode)方式下的中断和异常处理机制。参考第 15 章“调试和行为(性能)检测”中有关实地址和虚拟-8086 模式下中断和异常处理机制的区别的描述。

5.1.中断和异常概述

中断和异常是强制性的执行流的转移，从当前正在执行的程序或任务转移到一个特殊的称作句柄的例程或任务。当硬件发出信号时，便产生中断，中断的产生同正在执行的程序是异步的，即中断的产生是随机的。其用于处理处理器的外部事件，比如为外设服务的请求。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，包括保护异常，页错误，内部机器错误。P6 家族和 Pentium 处理器还允许当出现硬件错误和总线错误时产生硬件检测异常。

处理器的中断和异常处理机制使中断和异常的处理对于应用程序和操作系统或可执行程序来说是透明的。当处理器收到中断信号或检测到异常时，便挂起当前正在运行的进程或任务，而转去执行中断或异常处理例程。中断或异常处理例程执行完之后，处理器继续被中断的进程或任务。被中断的进程或任务继续执行，就像从未被打断过一样，只有两种情况例外：无法从发生的异常恢复，中断使当前的程序终止。

本章描述了处理器在保护模式下的中断和异常的处理机制。在本章的最后给出了异常和异常产生条件的详细描述。参考第 16 章，模拟 8086，以获得实地址和虚拟 8086 模式下的中断和异常的处理机制。

5.1.1.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

5.1.1.1.外部中断

外部中断是通过处理器的引脚接收的，也可以通过局部 APIC 串行总线接收。P6 家族或 Pentium 处理器上主要的中断引脚是连接到局部 APIC 的 LINT[1：0]两个引脚（参考 7.5，"高级可编程中断控制器"（Advanced Programmable Interrupt Controller（APIC）），当局部 APIC 被关闭时，这两个引脚被分别配置成 INTR 和 NMI 引脚。当信号由 INTR 引脚传递给处理器时，便发生了一个外部中断，处理器从系统总线读取由外部中断控制器（如 8259A，参考 5.2.，“异常和中断向量”）发来的中断向量号。若信号从 NMI 引脚传递进来，则发生的是一个不可屏蔽中断（NMI），其向量号为 2。

当 APIC 打开时，可通过 APIC 向量表对 LINT[1:0]引脚编程，使其和处理器的任意异常和中断向量绑定。

处理器局部 APIC 能够和系统上的 I/O APIC 相连。由 I/O APIC 引脚接收到的外部中断能通过 APIC 串行总线（PICD[1：0]引脚）传达到局部 APIC。I/O APIC 决定中断向量号并将其送往局部 APIC。当一个系统拥有多个处理器时，处理器之间也可通过 APIC 串行总线相互传递中断信号。

Intel486 和早期的 Pentium 处理器不具有芯片内建的局部 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，外中断由系统板上的中断控制器（8259A）产生，这些信号由 INTR 引脚传递给处理器。

处理器具有另外一些也可以产生处理器中断的引脚，但本章的讨论并不适用于这些中断的处理。这些引脚有：RESET#，FLUSH#，STPCLK#，SMI#，R/S#，和INIT#。Which of these pins are included on a particular Intel Architecture processor is implementation dependent.

这些引脚的功能在各自处理器的参考书中有描述。12 章也对 SMI#做了介绍。

5.1.1.2.可屏蔽硬件中断

任何通过 INTR 引脚或局部 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所有 Intel 架构定义的中断向量(0~255)；而通过局部 APIC 传递的部分只能使用 16~255 号向量。

使用 EFLAGS 寄存器的 IF 位就可以屏蔽全部可屏蔽硬件中断(参考 5.6.1.“屏蔽可屏蔽硬件中断”)。注意当 0 号中断到 15 号中断通过局部 APIC 传递时，APIC 会指出错误的向量号。(Note that when interrupts 0 through 15 are delivered through the local

APIC, the APIC indicates the receipt of an illegal vector.）

5.1.1.3.由软件产生的中断

将中断向量号作为 INT 指令的操作数即可通过 INT 指令在程序中产生中断。比如，指令 INT 35 即可调用第 35 号中断处理例程。

0 到 255 号中断均可使用 INT 指令调用。但是，当处理器预先定义好的 NMI 中断被这样调用时，处理器作出的响应与真正 NMI 中断发生时的响应并不一样。也就是说，执行 INT 2(NMI 的向量号)时，NMI 处理例程被调用，但是处理器的 NMI 硬件处理并未被激活。

注意：EFLAGS 的 IF 位并不能屏蔽由 INT 指令产生的中断。

5.1.2.异常源

处理器接收的异常信号有三个来源：

处理器检测到的程序错误异常

软件产生的异常

机器检测异常

5.1.2.1. 程序错误异常

在应用程序执行过程中，或操作系统执行中，当检测到程序错误时，处理器产生一个或多个异常。Intel 为每个处理器可检测到的异常定义了一个向量号。异常又进一步被划分为错误，陷阱和终止（参考 5.3.，“异常分类”）。

5.1.2.2. 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检测指定的异常条件。例如，INT 3 产生一个中断异常。

INT n 指令可以在软件中用来模拟某个异常，但有一点要注意。若该指令中的 n 指向 Intel 定义的某个异常，处理器便会产生一个指向相应的中断，接着就是调用相应的处理例程。这其实就相当于一个中断，处理器并不将出错码压入堆栈。于是，即便该异常原本有一个出错码，这时也被略去了。但对于那些带有出错码的异常处理例程，它们退出时会试图去弹出一个并不存在的出错码。此时，处理例程将 EIP 认为是错误码而弹出，而将一个无关的值弹出给 EIP，于是程序返回到一个错误的地方去了。

5.1.2.3. 机器检测异常

P6 系列和 Pentium 处理器同时提供了内部和外部的机器检测机制，用来检查内部芯片部件的操作和总线传输。这些机制组成了扩展异常机制（并不能独立完成）。当检测到一个机器

检测错误时，处理器发出一个机器检测异常（18 号向量），并返回一个出错码。参考本章后面的“18 号中断——机器检测异常（#MC）”和 13 章，机器检测结构。

5.2.异常和中断向量

处理器为每个异常和中断分配了一个识别码，称作向量。表 5-1 列出了异常和中断向量的分配情况，该表还提供了每个向量的异常类型，某个异常是否含有出错码，并给出了异常和中断源。

向量号从 0 到 31 被分配给异常和 NMI 中断使用。但目前的处理器还未使用完全部的这 32 个向量。未使用的向量号保留给将来使用。不要将其移做它用。

32 到 255 之间的向量号提供给用户使用。这些中断不在 Intel 的保留部分之列，一般被分配给外部 I/O 设备，允许它们通过某个外部硬件中断机制（5.1.1.,“中断源有叙述”）向处理器传递信号。

5.3.异常分类

在不失进程执行连续性的同时，按引起异常的指令能否重新执行，且依据它们被报告的方式，异常分为错误，陷阱和终止三种情况。

错误

错误是一种通常能够被修正的异常，一旦修正，程序能够不失连续性地接着执行。当报告错误发生时，处理器将机器状态恢复到执行错误之前的状态。错误处理例程的返回地址（CS 和 EIP 的存储值）指向产生错误的指令，而不是产生错误指令之后的那条指令。

注意：只有少数几个异常被报告为错误，but under architectural corner cases,
它们是不可恢复的，且处理器的上下文中的内容也会有部分丢失。一个例子：当执行 POPAD 指令是堆栈越过了堆栈段的尾部。异常处理例程会看到 CS：EIP 恢复原样，就好象 POPAD 从未执行，但处理器状态却被改变了（通用寄存器）。这种情况被视为程序错误，若应用程序产生这样的错误则会被操作系统终止。

陷阱

陷阱是一种异常，当引起陷阱的指令发生时，马上产生该异常。陷阱允许程序不失连续性的继续执行。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

终止是另一种异常，它并不总是报告产生异常的指令的确切位置，也不允许引起终止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，不一致或非法系统表值。

Table 5-1. Protected-Mode Exceptions and Interrupts

Vector No.	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug	Fault/ Trap	No	Any code or data reference or the INT 1 instruction.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (Zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	Floating-Point Error (Math Fault)	Fault	No	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[3]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[4]
19	#XF	Streaming SIMD Extensions	Fault	No	SIMD floating-point instructions[5]
20-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Nonreserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

NOTES:
1. The UD2 instruction was introduced in the Pentium® Pro processor.
2. Intel Architecture processors after the Intel386™ processor do not generate this exception.
3. This exception was introduced in the Intel486™ processor.
4. This exception was introduced in the Pentium® processor and enhanced in the P6 family processors.
5. This exception was introduced in the Pentium® III processor.

5.4.程序或任务的继续执行

为了使“从中断或异常处理例程返回的”被中断的程序或任务能继续执行，除“终止”之外的所有异常均严格地在前一条指令结束而下一条指令未开始执行时被报告，中断也是在该时刻被检测的。

对于错误类的异常，返回地址指向产生错误的指令。所以当从错误处理例程返回时，产生错误的指令将重新被执行。重新执行产生错误的指令通常用来处理当访问操作数受挫时的异常情况。最常见的例子是页错误异常（#PF），当某一进程或任务访问某一页中的操作数而该页并不在内存中时，将会发生这种异常。这时，异常处理例程在将所引用的页面加载到内存后，被中断的进程会从产生错误的指令处重新开始执行。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错误之前的状态，这就保证了出错指令的重新执行对被中断的进程或任务来说是透明的。

对陷阱类异常来说，返回地址指针指向的是产生陷阱指令的下一条指令。当一条转移指令执行过程中检测到陷阱时，返回地址指针则反映了执行转向的情况。例如，当执行JMP指令时，检测到有陷阱异常，返回地址指针指向的是JMP的目的地址，而不是JMP指令后的下一条指令。所有的陷阱异常保证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，返回地址指针指向的是INTO指令的下一条指令，该指令的作用是检查EFLAGS寄存器的OF位(溢出位)。该异常的陷阱处理例程解决(解析)了溢出条件。(The trap handler for this exception resolves the overflow condition.) 从陷阱处理例程返回时，进程或任务从INTO指令的下一条指令处开始执行。

终止类异常不支持进程或任务的继续执行。终止处理例程的作用是：当有终止异常发生时，收集处理器的各种相关诊断信息，并关闭进程或系统。

中断则绝对保证了在不失连续性的条件下，使被中断的进程和任务能继续执行。返回地址指针指向发生中断时的下一条指令。对于带重复前缀的指令，中断发生在两次循环之间。

5.5.不可屏蔽中断（NMI）

在两种情况下产生不可屏蔽中断（NMI）：

外部硬件向 NMI 引脚发信号

处理器从 APIC 串行总线上收到 NMI 模式的信号

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（参考 5.5.1.,“处理多个 NMI”）。

NMI 不能被 EFLAGS 的 IF 位屏蔽。

可以将一个可屏蔽硬件中断重定向到 2 号向量，以调用 NMI 处理例程；但是，这种中断不是真正的 NMI 中断。真正的 NMI 可以激活处理器的硬件处理部分，只能由上面提到的两种情况产生 NMI。

5.5.1.处理多个 NMI

当NMI处理例程执行时，处理器会禁止响应后继产生的NMI请求，知道有IRET指令执行。This blocking of subsequent NMIs prevents stacking up calls to the NMI handler.建议使用中断门来调用NMI中断处理例程，以屏蔽可屏蔽硬件中断（参考5.6.1.,“屏蔽可屏蔽硬件中断”）。

5.6.打开和关闭中断

根据处理器的状态和 EFLAGS 的 IF 位和 RF 位，处理器可以禁止某些中断的产生。详见下面的描述。

5.6.1.屏蔽可屏蔽硬件中断

5.6.　多个异常或中断时的优先关系

如果在指令边界有多个异常或中断发生，处理器将以预定的顺序来为它们提供服务。表 5-3 显示了各类异常和中断源的优先关系。

Table 5-2. SIMD Floating-Point Exceptions Priority

Priority	Description
1(Highest)	Invalid operation exception due to SNaN operand (or any NaN operand for max, min, or certain compare and convert operations)
2	QNaN operand[1]
3	Any other invalid operation exception not mentioned above or a divide-by-zero exception[2]
4	Denormal operand exception[2]
5	Numeric overflow and underflow exceptions possibly in conjunction with the inexact result exception[2]
6(Lowest)	Inexact result exception

5.8.中断描述符表（IDT）

中断描述符表（IDT）为每一个异常或中断向量对应的例程或任务分配了一个门描述符。同 GDT 和诸多 LDT 一样，IDT 也是由一系列由 8 个字节组成的描述符组成的（在保护模式下）。和 GDT 不同的是，IDT 中的第一个元不是 NULL 描述符。异常或中断向量号乘上 8 即可得到 IDT 中的描述符的索引（即门描述符包含的字节数）。由于只有 256 个中断或异常向量，所以 IDT 不必包含多于 256 个描述符。并且可以包含不足 256 个的描述符，因为只有那些确实发生的异常或中断才需要一个描述符。所有 IDT 中的空描述符须将存在位置位 0。

Table 5-3. Priority Among Simultaneous Exceptions and Interrupts

Priority	Descriptions
1 (Highest)	Hardware Reset and Machine Checks - RESET - Machine Check
2	Trap on Task Switch - T flag in TSS is set
3	External Hardware Interventions - FLUSH - STOPCLK - SMI - INIT
4	Traps on the Previous Instruction - Breakpoints - Debug Trap Exceptions (TF flag set or data/I-O breakpoint)
5	External Interrupts - NMI Interrupts - Maskable Hardware Interrupts
6	Faults from Fetching Next Instruction - Code Breakpoint Fault - Code-Segment Limit Violation[1] - Code Page Fault[1]
7	Faults from Decoding the Next Instruction - Instruction length > 15 bytes - Illegal Opcode - Coprocessor Not Available
8 (Lowest)	Faults on Executing an Instruction - Floating-point exception - Overflow - Bound error - Invalid TSS - Segment Not Present - Stack fault - General Protection - Data Page Fault - Alignment Check - SIMD floating-point exception

注释：

1. 对 Pentium 和 Intel486 处理器来说，代码段限长违规和代码页错误异常被赋以优先级 7。

The base addresses of the IDT should be aligned on an 8-byte boundary to maximize performance

of cache line fills. 限长以字节为单位，其与段基的和即为最后一个合法字节的地址。若限长为0，则合法字节只有一个。因为IDT总是包含8字节的描述符项，所以限长为8的倍数减一。

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位的基址和16位的限长。

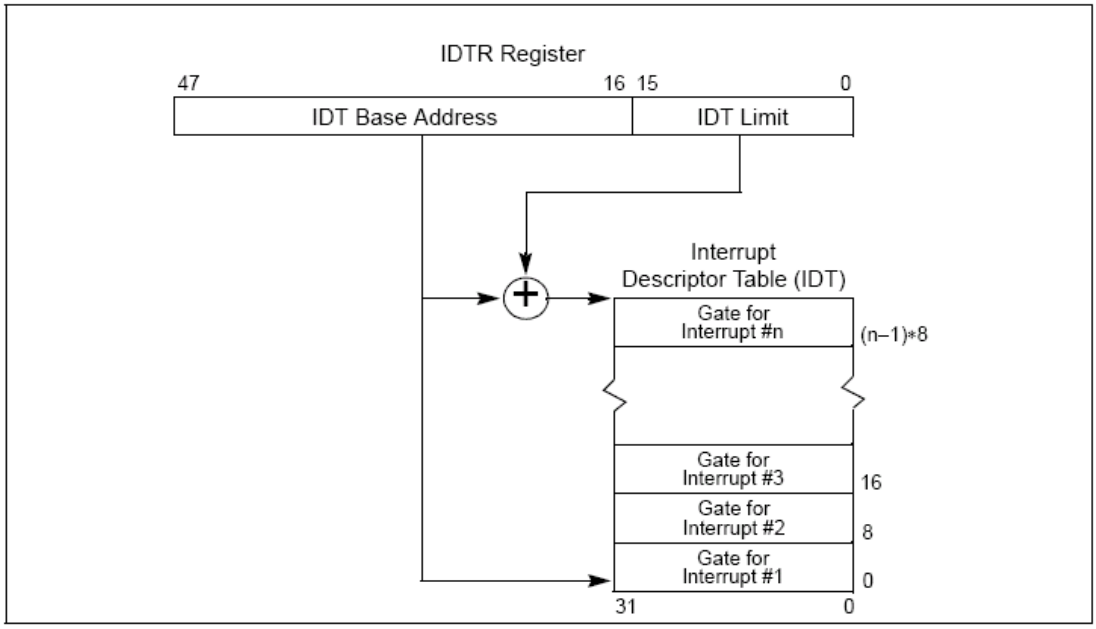

Figure 5-1. Relationship of the IDTR and IDT

LIDT和SIDT指令分别用来装载和保存IDTR寄存器的值。LIDT指令使用包含基址和限长的内存操作数装载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。SIDT指令将IDTR寄存器中的基址和限长保存到内存操作数中。可在任何特权级上使用。

如果引用的向量超过了IDT的限长，将发生通用保护错误（#GP）。

5.9.IDT 描述符

IDT 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 中使用的任务门的完全一样（参考 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 的段选择符。

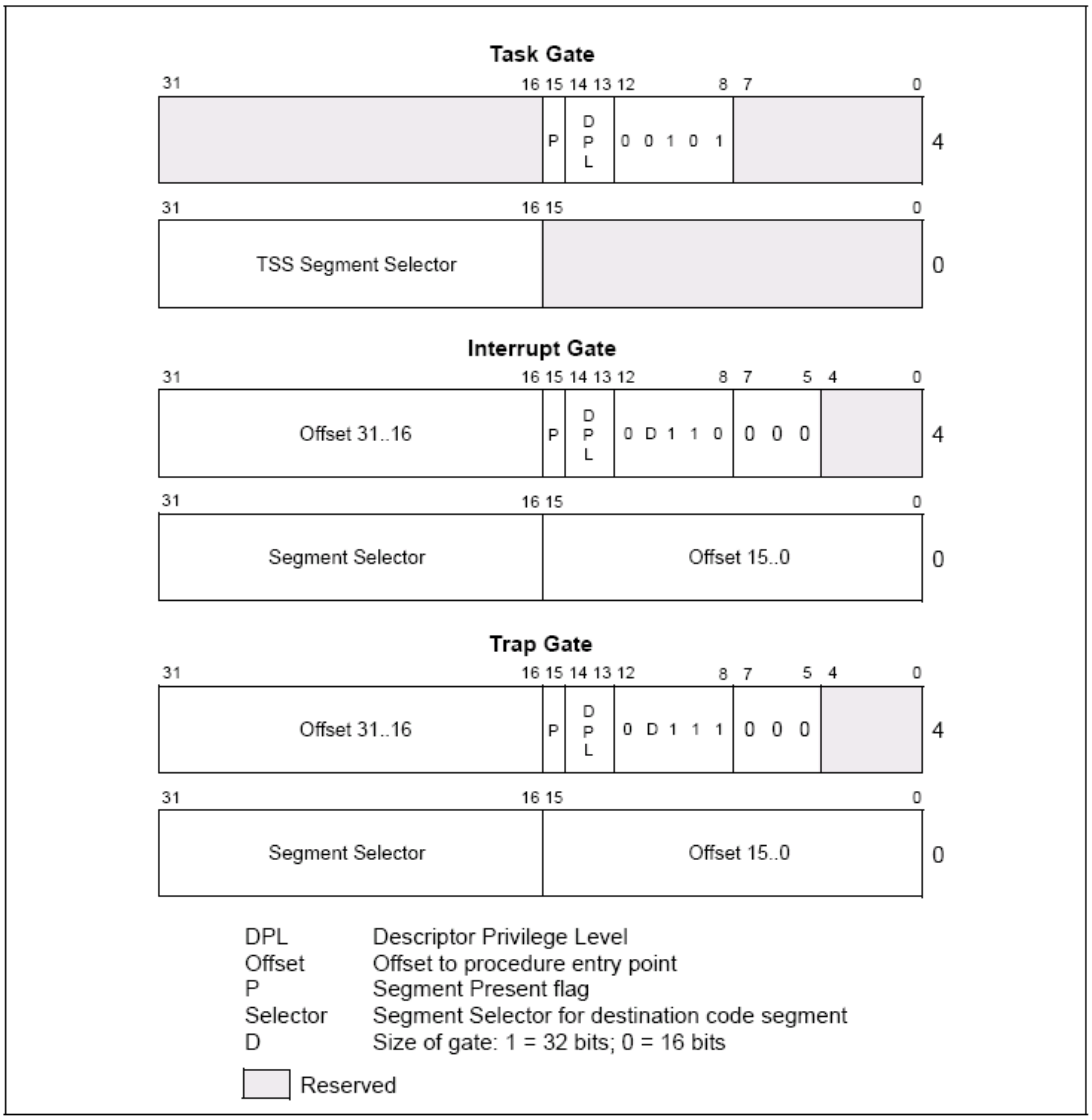

Figure 5-2. IDT Gate Descriptors

中断门和陷阱门同调用门（参考 4.8.3.,“调用门”）非常相似。它们包含一个远指针（段选择符和位移），处理器用其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 的 IF 位的方式上有所不同（参考 5.10.1.2.,“异常或中断对标志位的使用”）。

5.10.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近。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",一直到 4.8.6.,"从被调用例程返回")。若该索引指向的是任务门，

处理器会执行任务切换，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近(参考 6.3.,"任务切换")。

5.10.1 异常或中断处理例程

中断门或陷阱门引用一个异常或中断处理例程，这个例程运行于当前执行任务的上下文中(参考图 5-3)。门中的段选择符指向位于 GDT 或当前 LDT 中的可执行代码段的段描述符。门描述符中的偏移字段指向异常或中断处理例程的入口。

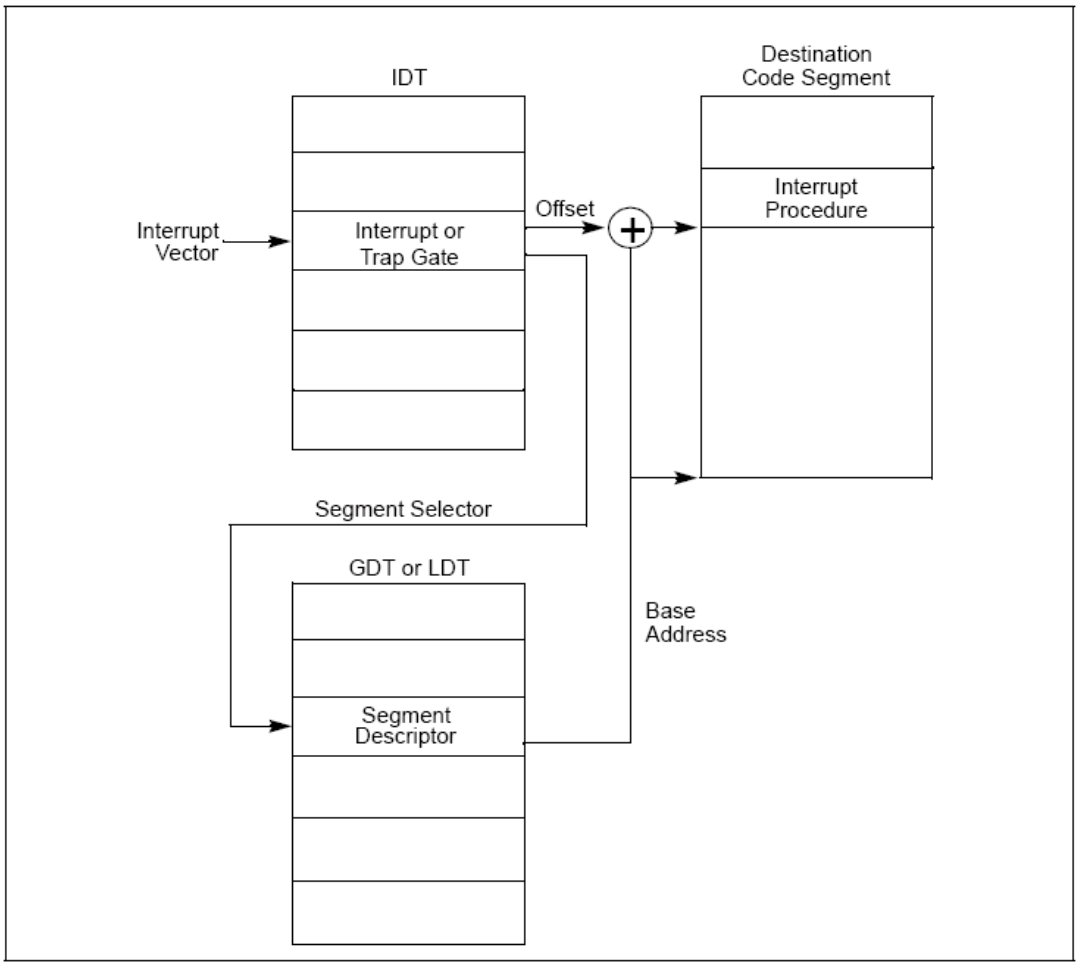

Figure 5-3. Interrupt Procedure Call

当处理器转去执行一个异常或中断处理例程时，会将 EFLAGS 寄存器，CS 寄存器，EIP 寄存器的当前值保存进栈(参考图 5-4)。(CS 和 EIP 寄存器为中断提供了一个返回地址指针。)如果异常同时产生了一个出错码，则该值也会压入栈中，位于 EIP 之后。[译者注：从图 5-4，可以看到，若未发生特权级的改变，被中断的进程和处理例程使用的是同一个堆栈，

即被中断进程的堆栈；而特权级发生改变时，则被中断的进程和处理例程将使用是不同的堆栈，而此时堆栈的指针由 TSS 中的相应字段给出。即处理例程使用的是被中断进程的高特权级堆栈，每个任务都有自己的独立的高特权级堆栈。这个问题在我学习保护模式的中断时一直困扰着我，故特别记在了这里。]

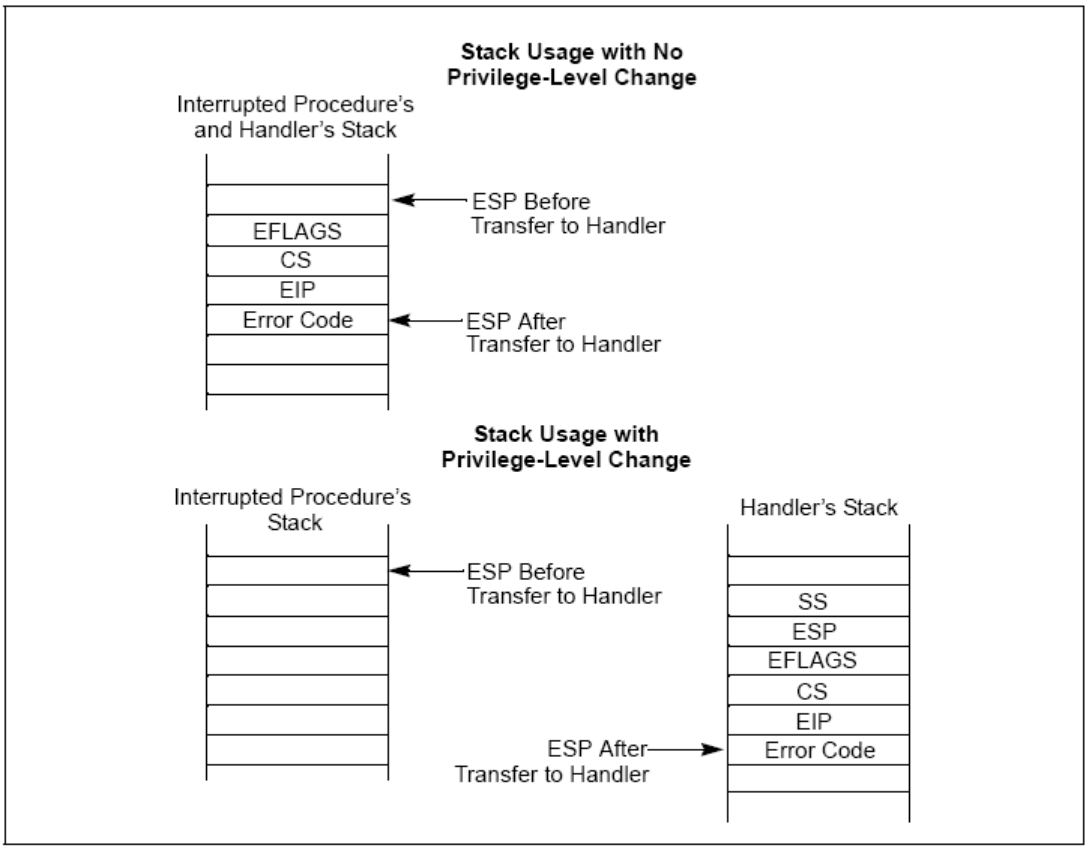

Figure 5-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

如果处理例程和被中断的进程处于同一特权级，则处理例程使用当前堆栈。

若当处理例程将运行于更高一级的特权级上时，堆栈发生切换。这时，指向返回后使用的栈指针也被压入栈中。(SS 和 ESP 用作处理例程返回后的栈指针。)而处理例程要使用的堆栈段选择符和栈指针则从当前进程的 TSS 中得到。处理器将 EFLAGS，SS，ESP，CS，EIP，还有出错码从当前进程的堆栈拷贝到处理例程的堆栈。

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标志位。只有当 CPL 为 0 时，EFLAGS 寄存器的 IOPL 位才恢复。IF 位只有在 CPL 小于或等于 IOPL 时才改变。参考第三章中“IRET/IRETD——中断返回”，以获取有关 IRET 指令的完整描述。

如果在调用处理例程时堆栈发生了切换，则在返回时，IRET 指令还将切换回被中断进程的堆栈。

5.10.1.1.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，同通过调用门的普通进程调用的特权级保护相似（参考 4.8.4.,“通过调用门访问代码段”）。如果异常和中断处理例程的特权级比 CPL 底，则处理器不允许这种调用发生。否则将产生通用保护异常（#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，所以当发生中断和异常时，并不检查 RPL。

仅当中断或异常由 INT n, INT 3，或 INTO 指令产生时，处理器才检查中断或陷阱门的 DPL。此时，CPL 必须小于或等于门的 DPL。这种限制防止了运行于 3 级的应用程序或进程使用软件中断来访问异常处理的关键代码，如页错误处理例程，因为这些例程位于更高一级的代码段中（数值上更小的特权级）。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中的 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加上了限制。下面提到的任一种技术都可避免特权级违例。

可以将异常或中断处理例程放到一致代码段中。这种技术只适用于仅访问堆栈上数据的处理例程（例如，除法错误异常）。如果该例程需要访问数据段中的数据，则此数据段必须能够被处在 3 级特权级的程序访问，这会导致数据无法处于保护之中。

可以将处理例程放到 0 特权级的非一致代码段中。则不管当前被中断进程或任务处于何级 CPL，处理例程总能够运行。

5.10.1.2.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 EFLAGS 寄存器的 TF 位。(当调用异常和中断处理例程时,处理器在将 EFLAGS 寄存器的内容保存进栈后，还会清 VM，RF，和 NT 位。）清 TF 位则可以禁止指令跟踪，以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 寄存器中的值，恢复 TF（和 VM，RF，及 NT）位。

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 位的方式。当通过中断门访问异常或中断处理例程时，处理器清除 IF 位，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 位不受影响。

5.10.2.中断任务

当异常或中断处理例程通过 IDT 中的任务门被访问时，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上下文被自动保存起来。

处理异常或中断时，允许处理任务使用新 TSS 的新的 0 特权级堆栈。若异常或中断发生时仍使用当前 0 特权级堆栈，则会搞糟堆栈，而通过任务门访问处理任务且使用新的 0 特权级堆栈可以防止系统崩溃。

通过给处理任务一个单独的地址空间，处理任务可以和其他任务隔离开来。这可由分配给它一个单独的 LDT 来实现。

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 中的任务门引用 GDT 中的某个 TSS 描述符（参考图 5-5）。切换到处理任务与普通的任务之间的切换完全一样（参考 6.3.，“任务切换”）。返回到被中断任务的链指针保存在处理任务 TSS 的链域字段中。如果异常导致了出错码，则出错码也被拷贝到了新任务的堆栈上。

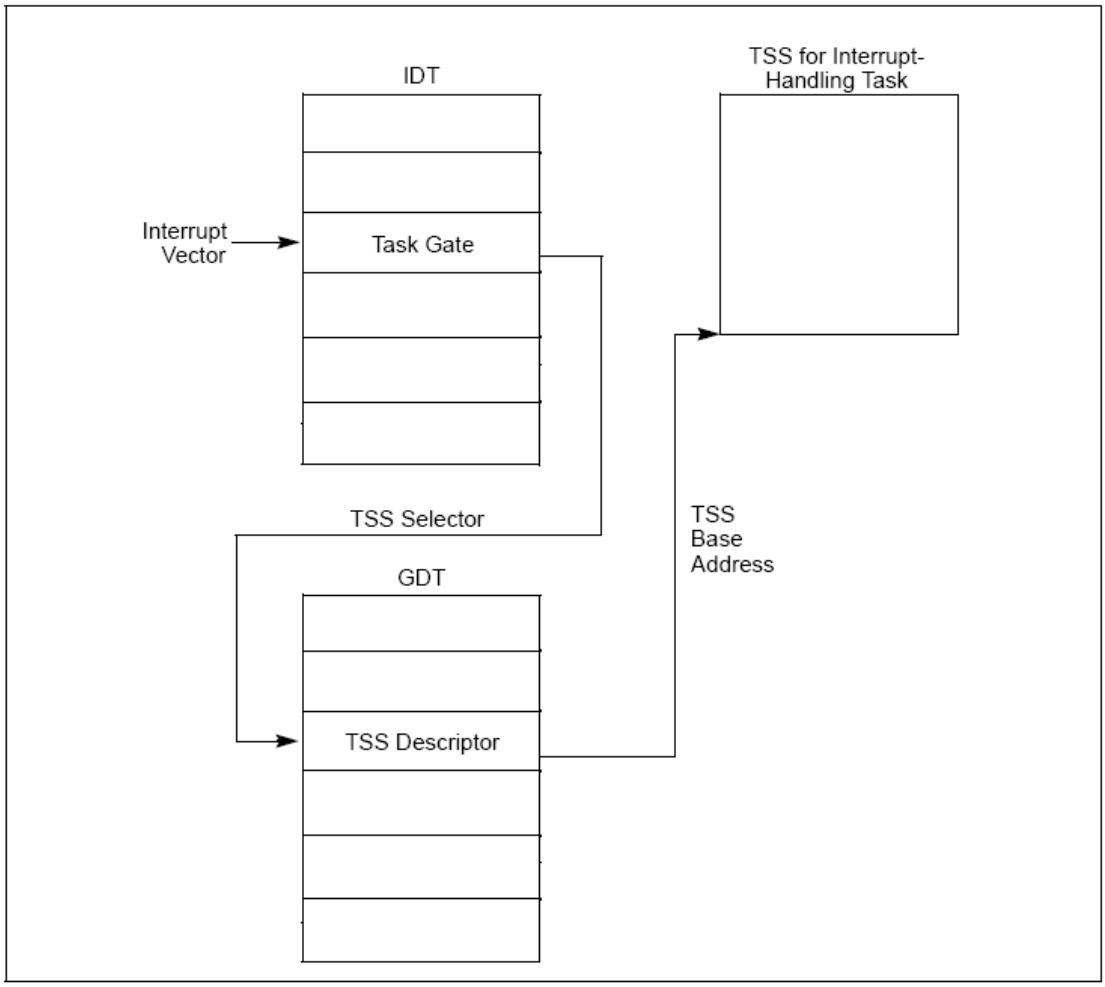

Figure 5-5. Interrupt Task Switch

当操作系统中使用异常或中断处理任务时，就有两种机制可用于调度任务：软件调度（操作系统的一部分）和硬件调度（处理器中断机制的一部分）。当允许中断时，软件必须提供中断发生时被调度使用的中断任务。

5.11.出错码

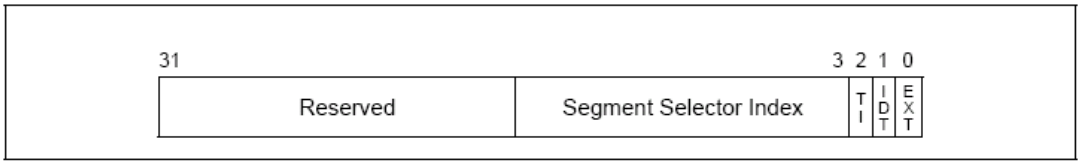

Figure 5-6. Error Code

Table 5-1. Protected-Mode Exceptions and Interrupts

Vector No.	Mne-monic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug	Fault/ Trap	No	Any code or data reference or the INT 1 instruction.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (Zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	Floating-Point Error (Math Fault)	Fault	No	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[3]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[4]
19	#XF	Streaming SIMD Extensions	Fault	No	SIMD floating-point instructions[5]
20-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Nonreserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

5.12.异常和中断参考

下面的内容产生异常和中断的条件。按照向量码的顺序排列。这里提到的内容包括：

异常类型

指出异常是错误，陷阱，还是终止。有些异常既可以是错误也可以是陷阱，这有赖于错误是何时被检测到的。（这部分对中断不适用。）

描述

给出了某类异常或中断使用目的的一般性的描述。它描述了处理器如何处理异常或中

断的。

异常出错码

指示是否为异常保存出错码。如果是，则将描述出错码的内容。（这部分对中断并不适用。）

被保存的指令指针

指出返回时指令指针指向哪条指令。它也指示了该指针是否被用于重新执行出错指令。

进程状态变化

描述了异常或中断对当前执行进程或任务产生的影响，和不是连续性的继续执行进程和任务的可能性。

0 号中断——除法错异常（#DE）

异常类型 错误

描述

第 7 章 多处理器管理

IA-32 体系提供了几种机制来管理和提升连接到同一系统总线的多个处理器的性能。这些机制包括：

- 总线加锁和 cache 一致性管理以实现对系统内存的原子操作。
- 串行化指令（serializing instructions.这些指令仅对 pentium4,Intel Xeon, P6, Pentium 处理器有效）
- 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 是在 Pentium 处理器中被引入 IA-32 体系的。
- 二级缓存（level 2， L2）。对于 Pentium4,Intel Xeon, P6 处理器，L2 cache 已经紧密的封装到了处理器中。而 Pentium, Intel486 提供了用于支持外部 L2 cache 的管脚。
- 超线程技术。这个技术是 IA-32 体系的扩展，它能够让一个处理器内核并发的执行两个或两个以上的指令流（参见 7.6，“超线程技术”）。

这些机制在对称多处理系统（symmetric-multiprocessing, SMP）中是极其有用的。然而，在一个 IA-32 处理器和一个专用处理器（例如通信、图形、视频处理器）共享系统总线的应用中，这些机制也是适用的。

这些多处理机制的设计目标是：

- 保持系统内存的完整性（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完整性，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——当一个处理器访问另一个处理器缓存中的数据时，必须要得到正确的数据。如果这个处理器修改了数据，那么所有的访问这个数据的处理器都要收到被修改后的数据。

- 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顺序必须要和编程时指定的写内存顺序相一致。
- 在一组处理器中派发中断处理——当几个处理器正在并行的工作在一个系统中时，有一个集中的机制是必要的，这个机制可以用来接收中断以及把他们派发到某一个适当的处理器。
- 采用现代操作系统和应用程序都具有的多线程和多进程的特性来提升系统的性能。

IA-32 体系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，AIPIC 体系在第 8 章，高级可编程中断控制器（Advanced Programmable Interrupt Controller, APIC）中讨论，总线和内存加锁，串行（serializing instructions）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（locked atomic operations）

32 位 IA-32 处理器支持对系统内存加锁的原子操作。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、系统段页表）。两个或多个处理器可能会同时的修改这些数据结构中的同一数据域或标志。处理器应用三个相互依赖的机制来实现加锁的原子操作：

- 可靠的原子操作（guaranteed atomic operations）。
- 总线加锁，使用 LOCK#信号和 LOCK 指令前缀。
- 缓存完整性协议，保证原子操作能够对缓存中的数据结构执行；这个机制出现在 Pentium 4，Intel Xeon，P6 系列处理器中。

这些机制以下面的形式相互依赖。某些基本的内存事务（memory transaction）（例如读写系统内存的一个字节）被保证是原子的。也就是说，一旦开始，处理器会保证这个操作会在另一个处理器或总线代理（bus agent）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），这些操作需要自动的处理，但又不能以上面的方式处理。因为频繁使用的内存数据经常被缓存在处理器的 L1、L2 高速缓存里，原子的操作通常是在处理器缓存内部进行的，并不需要声明总线加锁。这里的处理器缓存完整性协议保证了在缓冲内存上执行原子操作时其他缓存了相同内存区域的处理器被正确管理。

注意到这些处理加锁的原子操作的机制已经像 IA-32 处理器一样发展的越来越复杂。于是，最近的 IA-32 处理器（例如 Pentium 4, Intel Xeon, 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 可靠的原子操作

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总被自动的执行：

- 读或写一个字节
- 读或写一个在 16 位边界对齐的字
- 读或写一个在 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
总是被自动执行：

- 读或写一个再 64 位边界对齐的四字（quadword）
- 对 32 位数据总线可以容纳的未缓存的内存位置进行 16 位方式访问

（16-bit accesses to uncached memory locations that fit within a 32-bit data bus）
P6 系列处理器还保证下列内存操作被自动执行：

- 对 32 位缓冲线（cache line）可以容纳的缓存中的数据进行非对齐的 16 位，32 位，64 位访问

对于可以被缓存的但是却被总线宽度、缓冲线、页边界所分割的内存区域,Pentium 4, Intel Xeon, P6 family, Pentium 以及 Intel486 处理器都不保证访问操作是原子的。Pentium 4, Intel Xeon, P6 系列处理器提供了总线控制信号来允许外部的内存子系统完成对分割内存的原子性访问；但是，对于非对齐内存的访问会严重影响处理器的性能，因此应该尽量避免。

7.1.2 总线加锁（Bus Locking）

IA-32 处理器提供了 LOCK#信号。这个信号会在某些内存操作过程中被自动发出。当这个输出信号发出的时候，来自其他处理器或总线代理的总线控制请求将被阻塞。软件能够利用

在指令前面添加 LOCK 前缀来指定在其他情况下的也需要 LOCK 语义(LOCK semantics)。

在 Intel386, Intel486, Pentium 处理器中，直接调用加锁的指令会导致 LOCK#信号的产生。硬件的设计者需要保证系统硬件中 LOCK#信号的有效性，以控制多个处理对内存的访问。

对于 Pentium 4, Intel Xeon, 以及 P6 系列处理器，如果被访问的内存区域存在于处理器内部的高速缓存中，那么 LOCK#信号通常不被发出；但是处理器的缓存却要被锁定（参见 7.1.4, 加锁操作对于处理器内部缓存的影响)。

7.1.2.1自动加锁（Automatic Locking）

下面的操作会自动的带有 LOCK 语义：

- **执行引用内存的 XCHG 指令。**
- **设置 TSS 描述符的 B(busy 忙)标志。**在进行任务切换时，处理器检查并设置 TSS 描述符的 busy 标志。为了保证两个处理器不会同时切换到同一个任务。处理器会在检查和设置这个标志的时遵循 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，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，因此这个描述符不会在更新时被其他的处理器修改。为了使这个动作能够有效，更新描述符的操作系统过程应该采用下面的方法：
  - 使用加锁的操作修改访问权字节（access-rights byte)，来表明这个段描述符已经不存在，同时设置类型变量，表明这个描述符正在被更新。
  - 更新段描述符的内容。(这个操作可能需要多个内存访问;因此不能使用加锁指令。)
  - 使用加锁操作来修改访问权字节（access-rights byte)，来表明这个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（page-directory）和页表（page-table）的条目。**在更新页目录和页表的条目时，处理器使用加锁的周期（locked cycles）来设置访问标志和脏标志（dirty flag)。
- **响应中断。**发生中断后，中断控制器可能会使用数据总线给处理器传送中断向量。处

理器必须遵循 LOCK 语义来保证传送中断向量时数据总线上没有其他数据。

7.1.2.2软件控制的总线加锁

如果想强制执行 LOCK 语义，软件可以在下面的指令前使用 LOCK 前缀。当 LOCK 前缀被置于其他的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，一个非法操作码（invalid-opcode）异常会被抛出。

- 位测试和修改指令（BTS, BTR, BTC）
- 交换指令（XADD, CMPXCHG, CMPXCHG8B）
- XCHG 指令自动使用 LOCK 前缀
- 下列单操作数算术和逻辑指令：INC, DEC, NOT, NEG
- 下列双操作数算术和逻辑指令：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（一个用作处理器之间信号传递用的共享内存）。例如，如果一个处理器使用一个字来访问信号量，其他的处理器就不应该使用一个字节来访问这个信号量。

总线加锁的完整性不受内存区域对齐的影响。在所有更新操作数的总线周期内，加锁语义一直持续。但是建议加锁访问能够在自然边界对齐，这样可以提升系统性能：

- 任何边界的 8 位访问（加锁或不加锁）
- 16 位边界的加锁字访问。
- 32 位边界的加锁双字访问。
- 64 位边界的加锁四字访问。

对所有的内存操作和可见的外部事件来说，加锁的操作是原子的。只有取指令和页表操作能够越过加锁的指令。加锁的指令能用于同步数据，这个数据被一个处理器写而被其他处理器读。

对于 P6 系列处理器来说，加锁的操作使所有未完成的读写操作串行化（serialize）（也就

是等待它们执行完毕)。这条规则同样适用于 Pentium4 和 Intel Xeon 处理器，但有一个例外：对弱排序的内存类型的读入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

加锁的指令对于Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486处理器，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（self-modifying code）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(handling self- and cross-modifying code)

处理器将数据写入当前的代码段以实现将该数据作为代码来执行的目的，这个动作称为自修改代码。IA-32 处理器在执行自修改代码时采用特定模式的行为，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，而且可以在引退点（retirement point）之前推测性地执行接下来的代码（如：P4, Intel Xeon, P6 系列处理器)，如何判断应该执行哪段代码，是修改前地还是修改后的，就变得模糊不清。要想写出于现在的和将来的 IA-32 体系相兼容的自修改代码,必须选择下面的两种方式之一：
要想写出于现在的和将来的 IA-32 体系相兼容的自修改代码，必须选择下面的两种方式之一：

（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486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。）

需要注意的是自修改代码将会比非自修改代码的运行效率要低。性能损失的程度依赖于修改的频率以及代码本身的特性。

处理器将数据写入另外一个处理器的代码段以使得哪个处理器将该数据作为代码执行，这称为交叉修改代码（cross-modifying code)。像自修改代码一样，IA-32 处理器采用特定模式的行为执行交叉修改代码，具体依赖于被修改的代码与当前执行位置之间的距离。要想写出于现在的和将来的 IA-32 体系相兼容的自修改代码，下面的处理器同步算法必须被实现：

```
；修改的处理器
Memory_Flag ← 0; (* Set Memory_Flag to value other than 1 *)
将代码作为数据写入代码段;
Memory_Flag ← 1;

；执行的处理器
WHILE (Memory_Flag ≠ 1)
    等待代码更新;
ELIHW;
执行串行化指令; (* 例如, CPUID instruction *)
开始执行修改后的代码;
```

（在Pentium或486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。）

像自修改代码一样，交叉修改代码将会比非交叉修改代码的运行效率要低。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缓存的影响

对于Intel486和Pentium处理器，在进行加锁操作时，LOCK#信号总是在总线上发出，甚至锁定的内存区域已经缓存在处理器中。

对于Pentium 4, Intel Xeon, P6系列处理器，如果加锁的内存区域已经缓存在处理器中，处理器可能并不对总线发出LOCK#信号，而是仅仅修改缓存中的数据，然后依赖缓存一致性机制来保证加锁操作的自动执行。这个操作称为“缓存加锁”。缓存一致性机制会自动阻止两个或多个缓存了同一区域内存的处理器同时修改数据。

7.2 访存排序（memory ordering）

访存排序指的是处理器如何安排通过系统总线对系统内存访问的顺序。IA-32体系支持几种访存排序模型，具体依赖于体系的实现。例如，Intel386处理器强制执行“编程排序（program ordering）”（又称为强排序），在任何情况下，访存的顺序与它们出现在代码流中的顺序一致。

为了允许代码优化，IA-32体系在Pentium 4, Intel Xeon, P6系列处理器中允许强排序之外的另外一种模型——处理器排序（processor ordering）。这种排序模型允许读操作越过带缓存的写操作来提升性能。这个模型的目标是在多处理器系统中，在保持内存一致性的前提下，提高指令执行速度。

接下来的部分介绍Intel486和Pentium, Pentium 4, Intel Xeon, P6系列处理器使用的访存排序模型。

7.2.1 Pentium 和 Intel486 处理器的访存排序

Pentium和Intel486处理器遵循处理器排序访存模型；但是，在大多数情况下，访存操作还是强排序，读写操作都是以编程时指定的顺序出现在系统总线上。除了在下面的情况时，未命中的读操作可以越过带缓冲的写操作：当所有的带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。

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。

在“处理器排序”处理器（例如，Pentium 4, Intel Xeon, P6系列处理器）上运行的软件不能依赖Pentium或Intel486处理器的强排序。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序列化指令来完成的。（参见7.2.4加强和削弱访存排序模型）

7.2.2 Pentium 4, Intel Xeon, P6 系列处理器的访存排序

Pentium 4, Intel Xeon, P6系列处理器也是使用“处理器排序”的访存模型，这种模型可以被进一步定义为“带有存储缓冲转发的写排序”（write ordered with store-buffer forwarding）。这种模型有下面的特点：

在一个单处理器系统中，对于定义为回写可缓冲（write-back cacheable）的内存区域，下面的排序规则将被应用：

1. 读能够被任意顺序执行。
2. 读可以越过缓冲写，但是处理器必须保证数据完整性（self-consistent）。
3. 对内存的写操作总是以编程顺序执行，除非写操作执行了CLFUSH指令以及利用非瞬时的移动指令（MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, MOVNTPD）来执行流存储操作（streamint stores）。
4. 写可以被缓冲。
5. 写不能够预先执行；它们只能等到其他指令执行完毕。
6. 在处理器中，来自于缓冲写的数据可以直接被发送到正在等待的读操作。
7. 读写操作都不能跨越I/O指令，加锁指令，或者序列化指令。
8. 读操作不能越过LFENCE和MFENCE指令。
9. 写操作不能越过SFECE和MFENCE指令。

第二条规则允许一个读操作越过写操作。然而如果写操作和读操作都是访问同一个内存区域，那么处理器内部的监视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操作。

第六条规则构成了一个例外，否则整个模型就是一个写排序模型（write ordered model）。注意“带有存储缓冲转发的写排序”（在本节开始的时候介绍）指的是第2条规则和第6条规则的组合之后产生的效果。

在一个多处理器系统中，下面的排序规则将被应用：

- 每个处理器使用同单处理器系统一样的排序规则。
- 所有处理器所观察到的某个处理器的写操作顺序是相同的。
- 每个处理器的写操作并不与其它处理器之间进行排序。

图7-1解释了后面的规则。在一个三处理器的系统中，每个处理器执行三个写操作，分别对三个地址A, B, C。每个处理器以编程的顺序执行操作，但是由于总线仲裁和其他的内存访问机制，三个处理器执行写操作的顺序可能每次都不相同。最终的A, B, C的值会因每次执行的顺序而改变。

本节介绍的处理器排序模型与Pentium Intel486处理器使用的模型是一样的。唯一在Pentium 4, Intel Xeon, P6系列处理器中得到加强的是：

- 对于预先执行读操作的支持。
- 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。
- 对于长串的存储和移动的无次序操作(out-of-Order Stores)(参见，Pentium 4, Intel Xeon, P6处理器对于串操作的无次序存储）

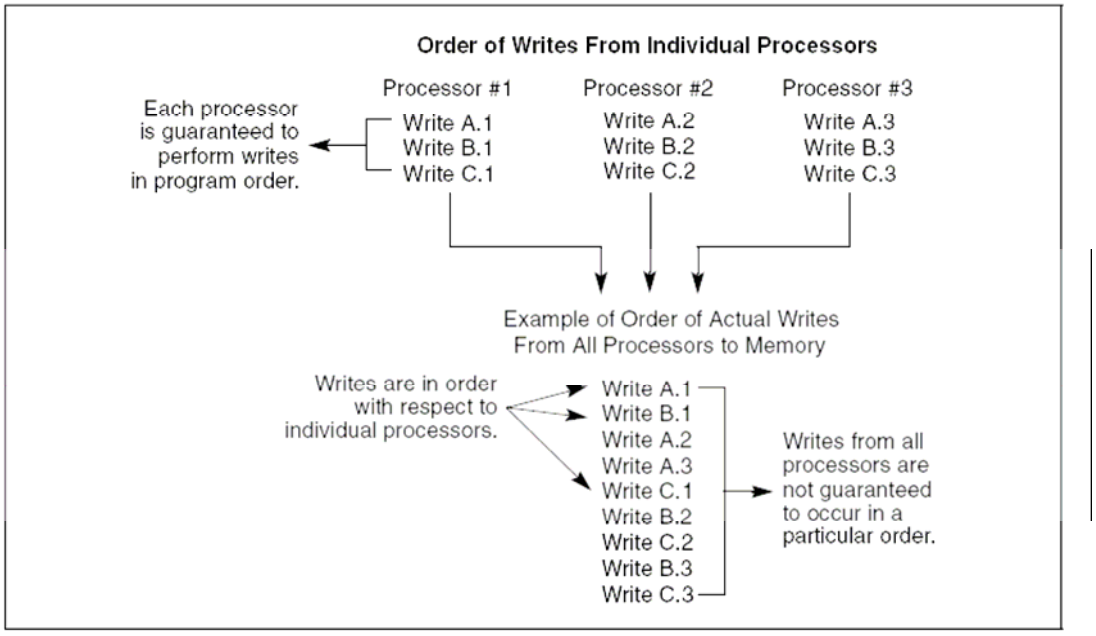

Figure 7-1. Example of Write Ordering in Multiple-Processor Systems

7.2.3 Pentium 4, Intel Xeon, P6 处理器对于串操作的无次序存储（Out-of-Order Stores）

Pentium 4, Intel Xeon, P6处理器在进行串存储的操作（以MOVS和STOS指令开始）时，修改了处理器的动作，以提升处理性能。一旦“快速串”的条件满足了（将在下面介绍），处理器将会在缓冲线（cache line）上以缓冲线模式进行操作。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相应中断。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。

按顺序存储串的代码不应该使用串操作指令。数据和信号量应该分开。依赖顺序的代码应该在每次串操作时使用信号量来保证存储数据的顺序在所有处理器看来是一致的。

“快速串”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Pentium4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。
- 源和目的内存的重合区域一定不能小于一个缓冲线的大小(Pentium 4和Intel Xeon 处理器是64字节；P6 和Pentium处理器是 32字节）。
- 源地址和目的地址的内存类型必须是WB或WC。

7.2.4 加强和削弱访存排序模型（Strengthening or Weakening the Memory Ordering Model）

IA-32体系提供了几种机制用来加强和削弱访存排序模型以处理特殊的编程场合。这些机制包括：

- I/O指令，加锁指令，LOCK前缀，以及序列化指令来强制执行“强排序”。
- 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（memory type range registers (MTRRs)）可以被用来加强和削弱物理内存中特定区域的访存排序模型。（参考，10.11“memory type range registers (MTRRs)”）。MTRRs只存在于Pentium 4, Intel Xeon, P6系列处理器。
- 页属性表可以被用来加强某个页或一组页的访存排序（参考10.12 “页属性表”Page Attribute Table(PAT)）。PAT只存在于Pentium 4, Intel Xeon, P6系列处理器。

这些机制可以通过下面的方式使用。

内存映射社和其他I/O设备通常对缓冲区写操作的顺序很敏感。I/O指令（IN,OUT）以下面的方式对这种访问执行强排序。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。只有取指令操作和页表查询（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。

一个多处理器的系统中的同步机制可能会依赖“强排序”模型。这里，一个程序使用加锁指令，例如XCHG或者LOCK前缀，来保证读-改-写操作是自动进行的。加锁操作像I/O指令一样等待所有之前的指令执行完毕以及缓冲区都被写入了内存（参考7.1.2 总线加锁）。

程序同步可以通过序列化指令来实现（参考7.4 序列化指令）。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上下文切换之前执行完毕。像I/O核加锁指令一样,处理器等待之前所有的指令执行完毕以及所有的缓冲区写入内存后才开始执行序列化指令。

SFENCE, LFENCE, MFENCE指令提供了高效的方式来保证读写内存的排序，这种操作发生在产生弱排序数据的程序和读取这个数据的程序之间。

- SFENCE——串行化发生在SFENCE指令之前的写操作但是不影响读操作。
- LFENCE——串行化发生在SFENCE指令之前的读操作但是不影响写操作。
- MFENCE——串行化发生在MFENCE指令之前的读写操作。

注意,SFENCE, LFENCE, MFENCE指令提供了比CPUID指令更灵活有效的控制内存排序的方式。

MTRRs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。下面的两个例子是利用MTRRs设置的内存类型如何来加强和削弱Pentium 4, Intel Xeon, P6系列处理器的访存排序：

- 强不可缓冲（strong uncached, UC）内存类型实行内存访问的强排序模型。这里，所有对UC内存区域的读写都出现在总线上，并且不能够被乱序或预先执行。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访存强排序。
- 对于可以容忍弱排序访问的内存区域，可以选择回写（write back, WB）内存类型。这里，读操作可以预先的被执行，写操作可以被缓冲和组合（combined）。对于这种类型的内存，锁定高速缓存是通过一个加锁的原子操作实现的，这个操作不会分割缓冲线，因此会减少典型的同步指令（如，XCHG在整个读-改-写操作周期要锁定数据总线）所带来的性能损失。对于WB内存，如果访问的数据已经存在于缓存中，XCHG指令会锁定高速缓存而不是数据总线。

PAT在Pentium III中引入，用来增强用于存储内存页的缓存性能。PAT机制通常被用来与MTRRs一起来加强页级别的高速缓存性能。表10-7显示了PAT与MTRRs的相互作用。

在Pentium 4, Intel Xeon, P6系列处理器上运行的软件最好假定是 “处理器排序”模型或者是更弱的访存排序模型。Pentium 4, Intel Xeon, P6系列处理器没有实现强访存排序模型，除了对于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，Intel并没有保证将来的处理器会支持这种模型。为了使软件兼容将来的处理器，操作系统最好提供临界区（critical region）和资源控制构建以及基于I/O，加锁，序列化指令的API，用于同步多处理器系统对共享内存区的访问。同时，软件不应该依赖处理器排序模型，因为也许系统硬件不支持这种访存模型。

7.3 向多个处理器广播页表和页目录条目的改变

在一个多处理器系统中，当一个处理器改变了一个页表或页目录的条目，这个改变必须要通知所有其它的处理器。这个过程通常称为“TLB shootdown”。广播页表或页目录条目的改变可以通过基于内存的信号量或者处理器间中断（interprocessor interrupts, IPI）。例如一个简单的，但是算法上是正确的TLB shootdown序列可能是下面的样子：

1. 开始屏障（begin barrier）——除了一个处理器外停止所有处理器；让他们执行HALT指令或者空循环。
2. 让那个没有停止的处理器改变PTE or PDE。
3. 让所有处理器在他们各自TLB中修改的PTE, PDE失效。
4. 结束屏障（end barrier）——恢复所有的处理器执行。

我们还可以针对性能优化一下TLB shootdown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。
- 操作系统可以处理在更新过程中处理器使用其他的（stale）映射的情况。

7.4 串行化指令（serializing instructions）

IA-32体系定义了几个串行化指令（SERIALIZING INSTRUCTIONS）。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，并且在执行下一条指令之前将所有缓冲区里的数据写入内存。例如：当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体系。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抑制指令的预执行（speculative execution），因为预执行的结果会被放弃掉。

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。
- 非特权访存排序指令——SFENCE, LFENCE, MFENCE。

当处理器执行串行化指令的时候，它保证在执行下一条指令之前，所有未完成的内存事务都被完成，包括写缓冲中的数据。任何指令不能越过串行化指令，串行化指令也不能越过其他指令（读，写， 取指令， I/O）。

CPUID指令可以在任何特权级下执行串行化操作而不影响程序执行流（program flow），除非EAX, EBX, ECX, EDX寄存器被修改了。

SFENCE, LFENCE, MFENCE指令为控制串行化读写内存提供了更多的粒度（参见7.2.4 加强和削弱访存排序模型）。

在使用串行化指令时，最好注意下面的额外信息：

- 处理器在执行串行化指令的时候并不将高速缓存中已经被修改的数据写回到内存中。软件可以通过WBINVD串行化指令强制修改的数据写回到内存中。但是频繁的使用WVINVD指令会严重的降低系统的性能。

- 当一条会影响分页设置（也就是改变了控制寄存器CR0的PG标志）的指令执行时，这条指令后面应该是一条跳转指令。跳转目标应该以新的PG标志（开启或关闭分页）来进行取指令操作，但跳转指令本身还是按先前的设置执行。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）。但是为了与其他IA-32处理器向前和向后兼容，最好是放置一条跳转指令。
- 在允许分页的情况下，当一条指令会改变CR3的内容时，下一条指令会根据新的CR3内容所设置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内容建立映射。（TLB中的全局条目并没有失效，参见10.9“Invalidating the Translation Lookaside Buffers(TLBs)”）。
- Pentium 4, Intel Xeon, P6系列以及Pentium处理器使用分支预测技术（branch-prediction）来提升系统性能——处理器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 多处理器初始化

IA-32体系（从P5系列处理器开始）定义了多处理器（MP）初始化协议，称为“多处理器规范1.4版”（Multiprocessor Specification Version 1.4）。这个浮泛定义了多处理器系统中的IA-32处理器的引导协议（这里，多处理器定义为两个或两个以上的处理器）。MP初始化协议有下面的特点：

- 不用专门的系统硬件就可以支持多处理器自举。
- 不需要专门的信号或者预先定义好的引导处理器，就可以初始化系统自举。
- 所有的IA-32处理器都采用相同的方式自举，包括那些支持超线程技术的处理器。

执行MP初始化协议的机制会因处理器型号而异，如下：

- P6系列的处理器——BSP和AP（参考7.5.1 BSP，AP处理器）的选择是在APIC总线上处理的，使用BIPI和FIPI消息。参见附录C，那里详细讨论了P6系列处理器的MP初始化。
- Intel Xeon处理器，family, model, stepping ID低于F09H的，BSP,AP的选择是通过在系统总线上的裁决来处理的，使用BIPI，FIPI消息。参考7.5.3，“Intel Xeon处理

器的MP初始化协议算法”，那里有详细的讨论。

- Intel Xeon处理器，family, model, stepping ID等于或高于F09H的，BSP,AP的选择是通过特殊的系统总线周期来完成的，并不需要使用BIPI，FIPI消息来进行裁决。这种选择方法在7.5.3中讨论，“Intel Xeon处理器的MP初始化协议算法”。

处理器的family, model, stepping ID可以通过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种类型的处理器：自举（bootstrap）处理器（BSP）和应用（application）处理器（AP）。在开启电源或RESET之后，系统硬件会动态的选择一个处理器作为BSP，余下的处理器作为AP。

作为BSP选择机制的一部分，BSP标志设置在IA32_APIC_BASE MSR中（参考图8-5），用以表明这个处理器是一个BSP。这个标志在其余的处理器中都被清除。

BSP执行了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。

在开启电源或RESET之后，AP完成一个小型的自我配置，然后等待BSP的启动信号(SIPI消息)。收到SIPI消息的时候，AP执行BIOS AP配置代码，结束HALT状态。

IA-32处理器支持超线程技术（Hyper-Threading），MP初始化协议将每个逻辑处理器都看作是一个总线上单独的处理器（带有唯一的APIC ID）。在启动的时候，一个逻辑处理器被选为BSP，另外的被选为AP。

7.5.2 MP 初始化协议的要求和 Intel Xeon 处理器的限制

MP初始化协议对系统有下列的需求和限制：

- MP协议仅仅在开启电源或RESET之后才执行。如果MP协议已经结束，BSP也已经被选定，随后的INIT（对某个特定的处理器或系统范围内）不会导致MP协议的再次执行。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BSP对AP发出INIT-SIPI-SIPI序列的时候以及AP对最后一个SIPI响应的时候。

7.5.3 Intel Xeon 处理器 MP 初始化协议算法

在开启电源或RESET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，下面的启动和初始化操作将被执行：

1. 根据系统拓扑结构，系统总线上的每个处理器都被赋予一个唯一的8bit APIC ID（参考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APICID寄存器中。
2. 根据APID ID，每个处理器都被分配一个判决优先级。
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来选择BSP和AP。BSP选择机制因family, model, stepping ID而不同：

   ——family, model, stepping ID等于或大于F0AH：

   - 所有处理器监听BNR#信号，这个信号不断的切换。当BSR#管脚停止切换的时候，每个处理器都尝试在总线上发送一个NOP特殊周期。
   - 带有最高判决优先级的处理器会成功的发送NOP周期，因此被选为BSP。该处理器设置自己的IA32_APIC_BASE MSR标志，然后从reset向量(物理地址 FFFF FFF0H)开始执行BIOS自举代码。
   - 余下的处理器（没有成功发出NOP周期的）被选为AP。它们保持BSP标志的清除状态并进入等待SIPI状态。

   ——family, model, stepping ID低于F0AH：

   - 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），选择自己为BSP并在IA32_APIC_BASE MSR中设置BSP标志（参考C.1 “P6系列处理器的MP初始化过程的概述”，那里详细讨论了

BIPI，FIPI，SIPI消息）

- 余下的处理器（没有被选为BSP的）被选为AP。它们保持BSP标志的清除状态并进入等待SIPI状态。
- 被选中的BSP对所有的处理器（包括自己）广播一个FIPI消息，这个消息是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reset向量(物理地址 FFFF FFF0H)开始执行BIOS自举代码。

5. 作为启动代码的一部分，BSP建立ACPI表和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设置处理器计数为1，然后对所有AP广播一个SIPI消息。这里，SIPI消息包含一个BIOS AP初始化向量地址（000VV000H，VV表示包含在SIPI中的向量）
7. AP初始化的第一步是对BIOS初始化信号量建立起一个竞争（在AP中）。第一个取得该信号量的AP开始执行初始化代码（参考7.5.4 “MP 初始化举例”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，并将处理器计数加1。执行了初始化代码后，AP执行CLI指令并进入HALT状态。
8. 当每个AP执行了AP初始化代码后，BSP就得到一个连接到系统总线上的处理器计数，之后BSP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HALT状态。在这个状态下，AP只能响应INIT, NMI, SMI。它们也可以响应STPCLK管脚的查询和设置（snoops and assertions）

下节将给出一个Intel Xeon 处理器MP初始化协议的例子。

附录B，特定模式寄存器（Model-specific Register），描述了MP初始化完成之后，怎样对本地APIC的LINT[0:1]管脚进行编程。

7.5.4 MP 初始化举例

下面的例子演示了在BSP和AP已经被选定后，如何利用MP初始化协议来初始化MP系统中的IA-32处理器。这段代码在运行在使用MP初始化协议的IA-32处理器，包括P6系列处理器，Pentium 4 处理器， Intel Xeon 处理器（有或没有Intel Hyper-Threading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。

```
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，参考7.5.3 “Intel Xeon处理器的MP初始化协议算法”)，BSP开始执行地址FFFF FFF0H的BIOS自举代码。自举代码通常完成下列操作：

1. 初始化内存
2. 装入微码更新
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“真正的Intel”
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地址空间被映射成强不可缓冲（strong unchaeable, UC）类型。
9. 从本地的APIC ID寄存器（默认为0）读取BSP的APIC ID：

   ```
   MOV ESI, APIC_ID      ; address of local APIC ID register
   MOV EAX, [ESI]
   AND EAX, 0FF000000H       ; zero out all other bits except APIC ID
   MOV BOOT_ID, EAX      ; save in memory
   ```

   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4K字节页中的AP启动代码的基地址转换为8-bit的向量。8bit向量定义了实地址模式（real-address mode）地址空间中的4K字节的页地址。例如，0BDH向量表示启动代码的地址为000BD000H。
11. 通过设置一个8bit的APIC虚拟向量寄存器（spurious vector register）开启本地APIC。

```
MOV ESI, SVR              ; address of SVR
MOV EAX, [ESI]
OR EAX, APIC_ENABLED      ; set bit 8 to enable (0 on reset)
MOV [ESI], EAX
```

12. 通过给APIC错误处理器设置一个8-bit向量来建立LVT错误处理入口。

```
MOV ESI, LVT3
MOV EAX, [ESI]
AND EAX, FFFFFF00H; clear out previous vector
OR EAX, 000000xxH; xx is the 8-bit vector the APIC error
                 ; handler.
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. 执行下列操作来检测AP的存在以及AP的个数：
    - 将COUNT的值设为1
    - 启动一个定时器（大概是100毫秒的长度）。在AP BIOS初始化代码中，AP将会增加COUNT变量。如果定时器到期了而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 IPI 序列，来唤醒AP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. 等待时间中断
17. 读取COUNT变量，建立处理器计数
18. 如果有必要，重新配置APIC并执行系统诊测。

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0H执行CPUID指令，然后读取EBX, ECX, EDX寄存器来决定AP是一个“真正的Intel”
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bit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INIT IPI

7.5.5 在 MP 系统中识别处理器

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 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（参考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是有8bit。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socket标识）。在将处理器配置为cluster的系统中，第3位和第4位构成两位的cluster ID。Intel Xeon MP中的第0位用来标识同一个物理封装中的两个逻辑处理器（参考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 MP中的一样。

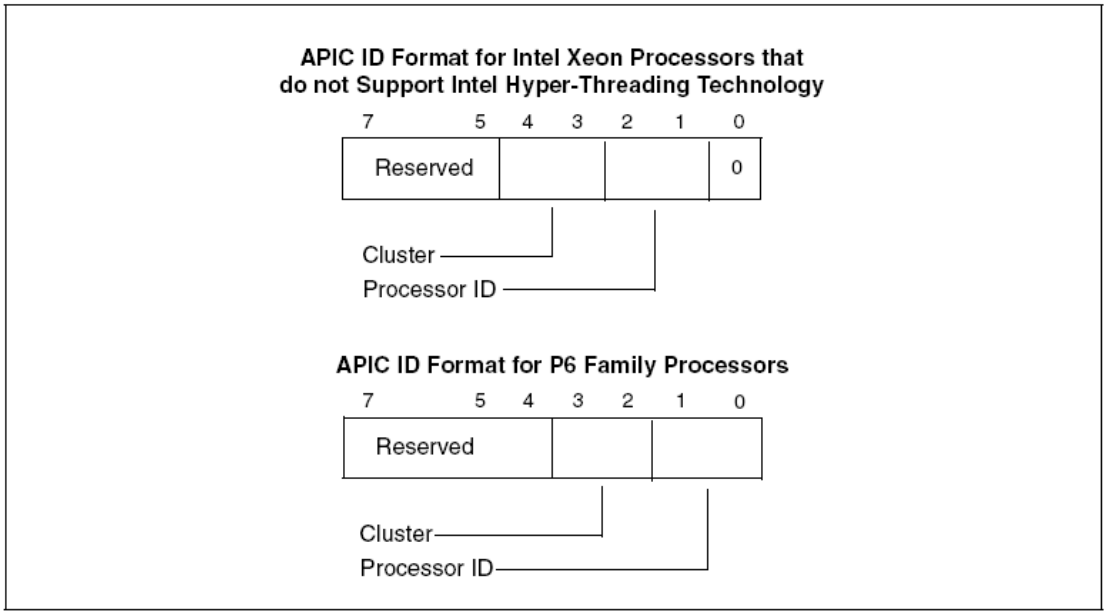

Figure 7-2. Interpretation of APIC ID in MP Systems

对于P6系列处理器，在启动和初始化时分配的APIC ID是有4bit（图7-2）这里第0、1位构成两位的处理器（或socket）标识。第2、3位构成两位cluster ID。

7.6 超线程技术

超线程技术在Intel Xeon处理器以及后续的处理器中被引入IA-32体系。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。参考 www.intel.com/info/hyperthreading 还有: 卷1, 2.2.4.,Hyper-Threading Technology.

Intel建议软件不要依赖IA-32处理器的名字来判断是否一个处理器支持超线程技术。软件应该使用CPUID指令，参考：7.6.3 “检测超线程技术”

超线程技术是对IA-32体系的一个扩展，它使一个物理处理器同时执行两个或更多代码流（称作线程,thread）。下面讨论IA-32如何实现这个技术。

7.6.1 Intel 超线程技术体系结构

图7-3 显示了支持超线程技术的IA-32处理器的大体结构,这里使用了Xeon处理器作为一个例子。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体系状态来表示）。它们共享一个处理器执行引擎和系统总线。每个逻辑处理器都有它自己的高级可编程中断控制器（APIC）。

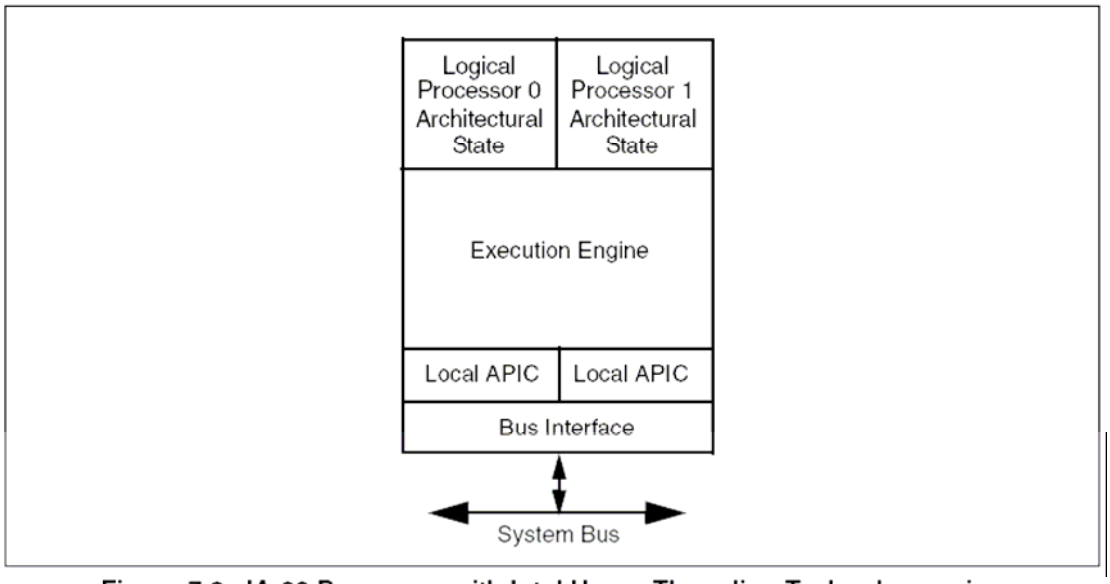

Figure 7-3. IA-32 Processor with Intel Hyper-Threading Technology using Two Logical Processors

7.6.1.1逻辑处理器的状态

下面的特性是支持超线程技术的IA-32处理器的逻辑处理器体系状态的一部分。这些特性被分成三组。

- 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

下面的特性需要每个逻辑处理器复制：

- 通用寄存器（EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP）
- 段寄存器(CS, DS, SS, ES, FS, and 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, task register)
- 调试寄存器(DR0, DR1, DR2, DR3, DR6, DR7)和调试控制MSRs
- 机器检查全局状态（Machine check global status ）(IA32_MCG_STATUS)和机器检查能力（ machine check capability）(IA32_MCG_CAP) MSRs
- 温度时钟调节(Thermal clock modulation)和ACPI电源管理控制MSRs
- 时间戳计数器（Time stamp counter ）MSRs
- 大部分其他MSR寄存器,包括页属性寄存器(PAT)。参考下面的例外部分。
- 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（Memory type range registers , MTRRs）

下面的特性是共享还是复制依赖于具体实现

- 机器检查体系（Machine check architecture , MCA）MSR(除了IA32_MCG_STATUS和IA32_MCG_CAP MSR)
- 性能监测控制和计数MSR(Performance monitoring control and counter MSRs)

7.6.1.2 APIC功能

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APIC ID（参考表8-1）。本地的APIC ID的作用就是标识逻辑处理器的ID，它被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，每个逻辑处理器都回被分配一个APIC ID（参考 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（interprocesor interrupt(IPI) message facility）来与逻辑处理器通信。无论处理器是否支持超线程技术，APIC的配置与编程都是一样的。参考第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（MEMORY TYPE RANGE REGISTERS，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，这个设置自动被同一个物理封装中的其他逻辑处理器共享。

IA-32体系要求所有使用IA-32处理器的MP系统必须使用相同的MTRR内存映射（memory map）。这样软件会得到一个统一的内存视图，而与它们正在那个处理器上运行无关。参考10.11，“内存范围寄存器（MEMORY TYPE RANGE REGISTERS，MTRR）”，关于如何设置MTRR。

7.6.1.4页属性表（PAGE ATTRIBUTE TABEL，PAT）

每个逻辑处理器都有一个单独的PAT MSR（IA32_CR_PAT）。但是，如10.12“PAGE ATTRIBUTE TABEL，PAT”中描述的，系统中所有处理器的PAT MSR设置必须一致，这也包括逻辑处理器。

7.6.1.5 机器检查体系（MACHINE CHECK ARCHITECTURE）

在超线程技术上下文内（HT Technology context），所有的机器检查体系（MACHINE CHECK ARCHITECTURE,MCA）MSR(除了IA32_MCG_STATUS, IA32_MCG_CAP MSR)都被每个逻辑处理器复制一份。这样在同一个物理封装内的逻辑处理器就可以同时的初始化、配置、查询以及处理异常。这个设计和第14章机器检查体系（MACHINE CHECK ARCHITECTURE）中介绍的机器检查异常处理是兼容的。

IA32_MCG_STATUS MSR是为每个逻辑处理器复制的，因此它的机器检查进行中（machine check in progress）标志（MCIP）可以用来检测MCA处理函数（MCA handlers）中的递归。另外，这个MSR允许每个逻辑处理器能够判断是否机器检查异常（machine-check exception）是否正在进行中，而不依赖于同一个物理封装中的其他逻辑处理器。

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查错误的通知。当一个严重错误发生时，如果机器检查异常被开启，所有在同一个物理处理器中的逻辑处理器都会运行及其检查异常处理程序。如果机器检查异常被关闭，逻辑处理器进入关闭状态并发出IERR#信号。

当开启机器检查异常时，CR4中的MCE标志应该在每个逻辑处理器中被设置。

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）以及它们自己的控制MSR。这些寄存器可以用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纪录（last branch record stack, LBR stack）堆栈。

7.6.1.7 性能监测计数器（PERFORMANCE MONITORING COUNTERS）

性能计数器（performance counters）和它们的控制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。性能计数中断（performance counter interrupts），事件，以及精确事件监测（precise event monitoring）可以在每个线程（逻辑处理器）中进行建立和分配。

参考15.11 “性能监测和超线程技术”，关于Intel Xeon处理器的性能监测的详细讨论。

7.6.1.8 IA32_MISC_ENABLE MSR

在支持超线程技术的IAIA32_MISC_ENABLE MSR(MSR地址1AH)是在各个逻辑处理器中共享的。这样，在同一个物理处理器中，这个寄存器所控制的特性对于各个逻辑处理器是相同的。

7.6.1.9访存排序（MEMORY ORDERING）

支持超线程技术的逻辑处理器和同不支持超线程技术的处理器遵循一样的访存排序（参考7.2 “访存排序”）每个逻辑处理器使用处理器排序访存模型（processor-order memory model），也就是带有存储缓冲的写排序（write-order with store buffer forwarding）。因特殊编程情况而对访存排序模型进行的加强和削弱对每个逻辑处理器都起作用。

7.6.1.10串行化指令

做为一条通用的规则，当一个逻辑处理器执行一条串行化指令的时候，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，WRMSR指令；在执

行MOV CR指令时，控制寄存器CR0中的CD标志被修改。这时候，所有逻辑处理器都被串行化了。

7.6.1.11微码更新资源（MICROCODE UPDATE RESOURCES）

在支持超线程技术的IA-32处理器中，微码更新功能是在逻辑处理器间共享的；每个逻辑处理器都能够开始一个更新。每个逻辑处理器都有它自己的BIOS签名MSR（BIOS signature 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（SELF MODIFYING CODE）

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。交叉修改也是被支持的，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。参考7.1.3，“处理自修改和交叉修改代码”，关于自修改和交叉修改的描述。

7.6.2 依赖实现的 HT 功能（Implementation-Specific HT Technology Facilities）

下面的功能是基于实现而决定的：

- 高速缓存
- 转换后备缓冲区（Translation lookaside buffer(TLB)）
- 温度监控功能

Intel Xeon 处理器MP的实现在后面的小节中讨论。

7.6.2.1处理器高速缓存（PROCESSOR CACHES）

对于Intel Xeon处理器MP，高速缓存是共享的。在逻辑处理器运行的任何缓存控制指令都会在物理处理器中的缓存层次体系产生全局影响。注意下面：

- WBINVD指令。在将数据写到内存后，整个缓存中的数据置为失效。所有的逻辑处理器都停止执行，直到写内存和失效操作执行完毕。一个特殊的总线cycle被发送到所有的缓存代理（caching agents）。
- INVD 指令。整个缓存中的数据被置为失效，但是并不需要写到内存中。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cycle被发送到所有的缓存代理（caching agents）。
- CLFUSH指令。指定的缓存单元（cache line）被置为失效，修改过的数据被写回内存，一个特殊的总线cycle被发送到所有的缓存代理（caching agents），无论哪个逻辑处理器导致缓存单元失效。

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标志是需要进行或运算的，因此其中一个逻辑处理器设置了CD标志，整个缓存就被关闭了。

7.6.2.2处理器转换后备缓冲区（Translation lookaside buffer(TLB)）

在Intel Xeon处理器MP中，数据缓存TLB是共享的，指令缓存TLB是每个逻辑处理器保存一个副本的。

TLB中的条目被用一个ID作了标记（tag），这个标记表示哪个逻辑处理器初始化了转换（translation）。甚至那些利用页全局属性标记为全局的转换也会用到这个标记。

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条目才会被flush。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的写操作以及使用INVLPG指令。

7.6.2.3温度监控

Intel Xeon MP中，逻辑处理器共享紧急关闭探测器（catastrophic shutdown detector）和自动温度监控机制（参考13.16 “温度监控和保护”）。共享会导致如下效果：

- 如果处理器的内核温度超过了预设的紧急关闭温度，处理器内核会停止执行，也就是所有的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紧急关闭温度，内核的时钟频率会自动地调整，所有的逻辑处理器都会受到影响。

对于控制时钟调整的软件，每个逻辑处理器都有它自己的IA32_CLOCK_MODULATION MSR，这会使每个处理器单独的控制始终调整的开启或关闭。如果希望软件来调整始终频率，一个物理处理器中的所有逻辑处理器都必须开启这个功能，并且设置duty cycle为相同的值，如果duty cycle是不同的。处理器时钟会选择最高的duty cycle。

7.6.2.4外部信号协调性（EXTERNAL SIGNAL COMPATIBILITY）

本节讨论对Intel Xeon MP管脚所收到的外部信号的限制以及这些信号是如何在逻辑处理器之中共享的。

- STPCLK#。Intel Xeon MP提供一个单独的STPCLK#管脚。外部的控制逻辑利用这个管脚来控制系统的电源管理。当STPCLK#有信号时，处理器内核转换到stop-grant状态，这个状态下，处理器停止执行，但是会继续响应探听事务（snoop transactions）.在STPCLK#信号到达时，无论逻辑处理器在运行或者停止，所有的逻辑处理器都会停止执行，并不再响应中断。

  在MP系统中，所有物理处理器的STPCLK#管脚都被绑定到一起。因此这个信号会同时影响系统中的所有逻辑处理器。

- LINT0和LINT1管脚。一个Intel Xeon MP只有一组LINT0和LINT1管脚，它们被逻辑处理器所共享。当这些管脚中有一个接收到信号，所有的逻辑处理器都会响应，除非这个管脚在一个或全部的逻辑处理器上被APIC本地向量表所屏蔽。

  在MP系统中，LINT0和LINT1管脚不是用来给逻辑处理器发送中断的。所有的中断都是通过I/O APIC传递给本地处理器的。

- A20M# Pin。在IA-32处理器上，A20M#管脚是用来与Intel 286处理器兼容的。这个管脚有信号时会导致物理地址的第20位对外部总线的内存访问被屏蔽（强制设为0）。Intel Xeon MP提供一个A20M#管脚，这个管脚会影响一个物理处理器中的所有逻辑处理器。这个配置与IA-32体系的是兼容的。

7.6.3 检测超线程技术

软件可以利用CPUID指令来检测一个IA-32处理器是否支持超线程技术以及处理器的配置情况。以EAX为1执行CPUID指令，下面的两个条目报告了超线程技术的情况：

- 超线程特征标志（EDX寄存器第28位）表明了（如果该标志被设置的话）处理器支持超线程技术。
- EBX的第16位到23位表明了一个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检测是否支持MONITOR/MWAIT指令

streaming SIMD Extensions 3引入了两个指令（MONITOR 和 MWAIT）来协助多线程软件提升线程的同步性。在开始的实现中，MONITOR和MWAIT对ring0的软件有效。对于运行级大于0的是有条件的支持。使用下面的方法来检测MONITOR/MWAIT指令的有效性：

- 用1H调用CPUID来查询MONITOR有效位（ECX[3]）。如果该位为1，那么ring0就支持MONITOR和MWAIT。
- 如果CPUID报告ECX[3]=1，在一个TRY/EXCEPT异常处理中运行MONITOR指令。如果会导致运行异常，说明在高于0的运行级上不支持MONITOR和MWAIT。参见例7-1。

Example 7-1. Verifying MONITOR/MWAIT Support

```
boolean MONITOR_MWAIT_works = TRUE;
try {
    _asm {
        xor ecx, ecx
        xor edx, edx
        mov eax, MemArea
        monitor
        }
    // Use monitor
} except (UNWIND) {
        // if we get here, MONITOR/MWAIT is not supported
    MONITOR_MWAIT_works = FALSE;
}
```

7.6.4 初始化支持超线程技术的 IA-32 处理器

在MP系统中初始化支持超线程技术的IA-32处理器的过程与一般的MP系统是一致的（参考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其他的处理器（或逻辑处理器）被选为AP。初始化过程同7.5.3“Intel Xeon 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，它将被存放在每个逻辑处理器的本地APIC ID寄存器中。如果两个或更多的处理器支持超线程技术，每个逻辑处理器都被分配一个唯一的ID（参考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，软件就可以通过发送APIC IPI消息来同逻辑处理器通信了。

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，其他的逻辑处理器被设置为HALT状态。要想让处于HALT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（interprocessor interrupt,IPI）。作为对该中断的响应，HALT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active的线程。共享的执行资源按照“按需分配”（as needed basis）原则分配给active的逻辑处理。

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）或者其他负载均衡机制来周期性的中断每个active的逻辑处理器。中断时，操作系统检查自己的运行队列并且取出一个线程派发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。

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，然后派发给特定的逻辑处理器（图7-4）。每个逻辑处理器都能够通过写本地APIC中的ICR寄存器的方式给其他的逻辑处理器发送IPI（参考8.6“发送处理器间中断”）。

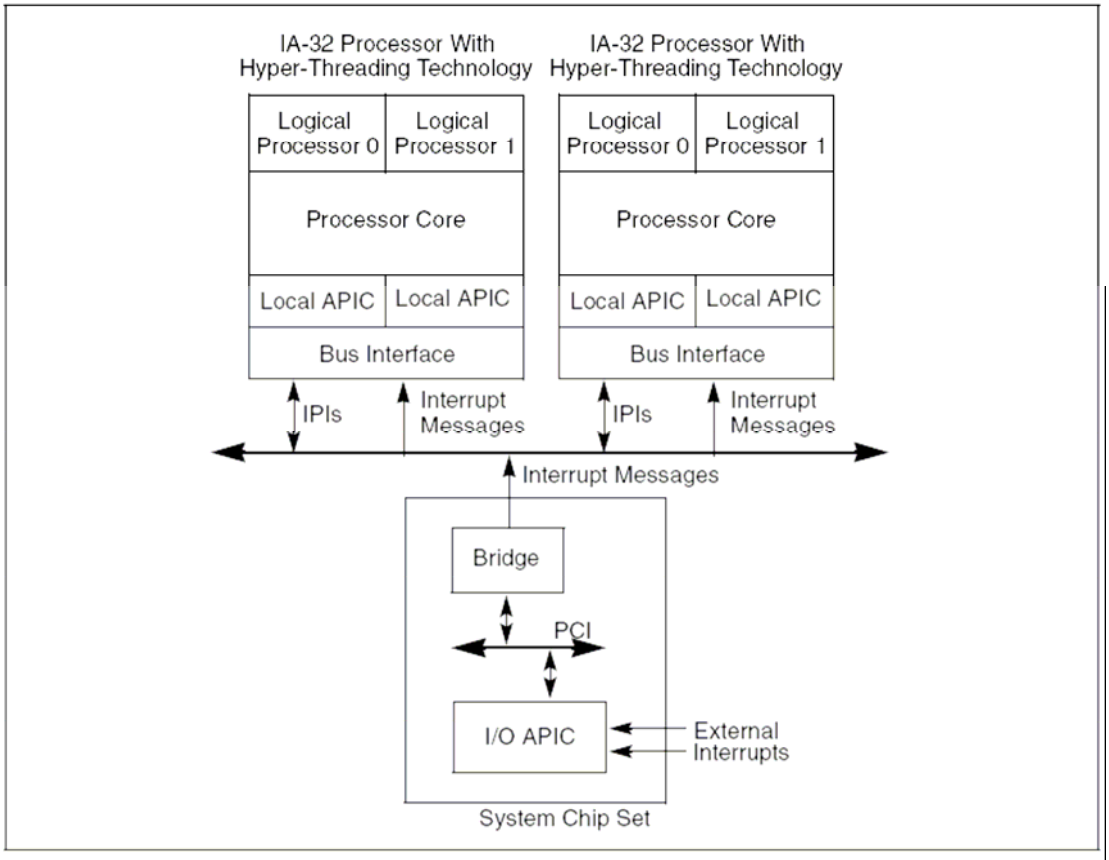

Figure 7-4. Local APICs and I/O APIC When IA-32 Processors Supporting Hyper-Threading Technology Are Used in MP Systems

7.7 空闲和阻塞情况的管理（MANAGEMENT OF IDLE AND BLOCKED CONDITIONS）

在一个逻辑处理器执行一个线程时，逻辑处理器可能会依照“按需分配”（on an as-needed basis）来使用共享的处理器资源（cache lines, TLB entries, bus accesses）。当一个

处理器处于空闲（没有工作可做）或阻塞（因为锁或信号量）的状态时，可以通过HLT（停止），PAUSE, MONITOR/MWAIT指令来对核心执行引擎进行进一步的管理。

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halted状态直到新的通知到达（until further notice）（参考IA-32 Intel Architecture Software Developer’s Manual , Volume 2, 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halted处理器使用的共享资源对active处理器来说都是可用的，这样就大大提高了执行效率。当halted处理器恢复执行时，共享资源又在所有的active处理器之间共享（参考 7.7.6.2 “停止空闲的逻辑处理器”，关于如何在超线程IA-32处理器上使用HLT指令）

7.7.2 PAUSE 指令

在执行“spin-wait loops”或者利用空循环（tight polling loop）其他访问共享锁或信号量时，PAUSE指令可以提高超线程IA-32处理器的性能。在执行spin-wait loop时，处理器的性能会大大的降低，因为在退出循环的时候，处理器会检测到访存排序违例（memory order violation），于是将内核流水线中的数据清空（flush）。PAUSE指令可以给处理器一个暗示，表示该代码序列是一个spin-wait loop。处理器可以利用这一点来避免访存排序违例，进而避免流水线flush。另外，PAUSE指令会de-pipeline spin-wait loop，使其不会消耗过多的执行资源（参考 7.7.6.2 “停止空闲的处理器”，关于HLT指令的更多介绍）

7.7.3　MONITOR/MWAIT 指令

操作系统通常需要空闲循环（idle loops）来处理线程同步。在典型的idle-loop中，可能存在忙循环（busy loops）并且这些循环会访问某些特定区域的内存。相关的处理器会在一个循环中等待某个内存区域来决定是否进行下一步的工作。接下来的工作一般是写内存

操作。初始化一项操作并且等待调度需要好几个总线周期。

从资源共享的角度来看（逻辑处理器共享执行资源），OS idle loop可以使用HLT指令，但是会有一些问题。执行HLT指令的逻辑处理器会处于停止状态。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（inter-processor interrupt）来唤醒停止中的处理器。这种中断的发出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会导致程序从busy loop中退出；这种状态的变化通常是由另一个代理（agent）（比如说另一个逻辑处理器）对该内存区域的写操作引起的。

MONITOR/MWAIT是对HLT和PAUSE的一个补充,目的是使共享同一物理资源的逻辑处理器对共享资源进行有效的分割或合并（patitioning and un-partitioning）。MONITOR建立起一个用来对写操作进行监视的内存地址范围；MWAIT将处理器置于optimized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视的内存范围。

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监视硬件的状态。监视硬件可以是armed(执行了MONITOR指令)或者是triggered（由于各种不同的事件引起，包括对被监视的内存区域进行写操作）。如果在执行MWAIT时，监视硬件处于triggered状态：MWAIT就像一条NOP指令一样并继续执行下面的指令。没有通用的方法可以得到监视硬件的状态，除非通过MWAIT的行为。

除了对被监视内存进行写操作之外,还有其他一些事件会导致执行MWAIT指令的处理器被唤醒。这些事件有些可以导致自愿或被迫的（voluntary or involuntary）上下文切换，例如：

- 外部的中断，包括NMI，SMI，INIT，BINIT，MCERR，A20M#
- 故障（fault），异常中止(abort)(包括机器检查（Machine Check））
- TLB失效操作，包括对CR0，CR3，CR4和某些MSR的写操作；执行LMSW（在设置了监视范围之后，但在发出MWAIT之前）
- 由fast system call或者far call产生的voluntary transitions

与电源管理相关的事件（例如Thermal Monitor 2或者chipset driven STPCLK#）和故障（fault）不会引起监视事件的进行状态被清除。

软件不应该有意的（voluntary）在MONITOR/MWAIT之间进行上下文切换。注意，执行MWAIT

不会重新arm监视硬件。也就是说MONITOR/MWAIT需要在一个循环中执行。还要注意的是从MWAIT退出可能是由于其他事件而不是对监视内存范围的写操作;软件应该自己检查一下监视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)检查监视区域的地址，以确定在执行MONITOR指令期间有数据被写入监视区域。

提供给MONITOR的监视区域必须是WRITE-BACK caching类型的。只有write-back类型的内存被写入时，才能触发监视硬件。如果监视区域的内存类型不是write-back的，地址监视硬件可能会被不正常的设置或者可能不会切换到arm状态。软件还应该保证下面的条件：

- 不想引起busy loop退出的写操作不会在监视区域中执行，
- 想引起busy loop退出的写操作一定要在监视区域中执行。

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既提供了判定监视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，软件应该了解监视区域的大小以及在多处理器系统中用于cache-snoop traffic的coherence line的大小（the size of the coherence line size for cache-snoop traffic in a multiprocessor system.）这个信息可以通过CPUID的monitor leaf功能（EAX=05H）。你可能会需要最大和最小monitor line的大小。

- 为了避免错过唤醒：确定被监视数据结构的大小要比最小monitor line-size要小。否则，处理器可能会错过向要唤醒MWAIT的写操作。
- 为了避免错误的唤醒：使用最大monitor line size来扩充（pad）所要监视的数据结构。软件必须保证在数据结构之外没有无关的

数据存在于监视区域。可能需要一个pad来避免这种情况。

上面的两个值和cache line大小是无关的，软件不应该假定它们之间有任何关系。在一个single-cluster系统中，最大和最小monitor line-size的数值是相等的。

基于CPUID返回的monitor line size，OS应该能够动态的分配带有适当填充（padding）的

数据结构。如果OS要使用一个静态的数据结构，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呈同步。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。

在multi-clustered系统上正确的为MONITOR/MWAIT建立数据结构：需要处理器、芯片组、BIOS间的交互（系统coherence line size可能会与芯片组有关；这个size可能会与监视区域的大小不同）。BIOS的职责是利用IA32_MONITOR_FILTER_LINE_SIZE MSR设置正确的coherence line size。监视区域的大小和IA32_MONITOR_FILTER_LINE_SIZE MSR的值经过比较后，较小的那个会被报告为最小Monitor Line Size，较大的被报告为Largest Monitor Line Size。

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power-up或RESET时会建立一个初始的APIC ID（参考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（physical package ID）以及cluster ID。图7-5显示了这些域的结构。这里，第0位是一个1bit的逻辑处理器ID，第1位和第2位组成一个2-bit的封装（package ID），第3、4位组成一个2-bit的cluster ID。第0位用来识别同一个封装中的两个逻辑处理器。

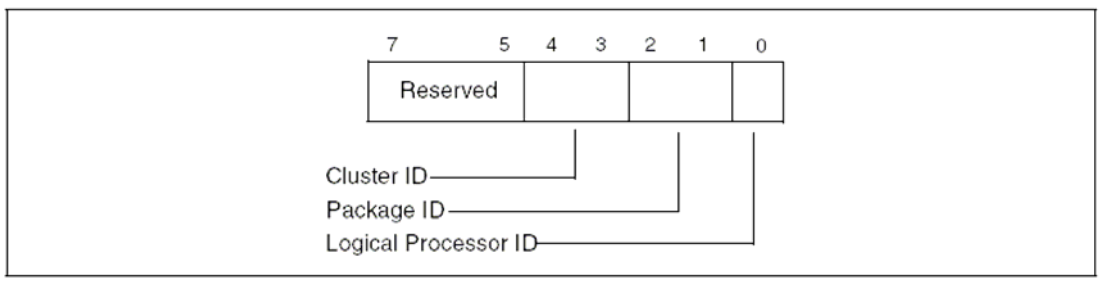

Figure 7-5. Interpretation of the APIC ID

表7-1显示了在一个有四个Intel Xeon MP处理器（总共有8个逻辑处理器）的系统中为每个逻辑处理器产生的APIC ID。一个Intel Xeon MP处理器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（primary logical processor），逻辑处理器1被称为“副逻辑处

理器（secondary logical processor）”。

Table 7-1. Initial APIC IDs for the Logical Processors in a System that has Four MP-Type Intel Xeon Processors Supporting Hyper-Threading Technology

Logical Processor Initial APIC ID	Physical Processor ID	Logical Processor 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3bit。package ID和cluster ID将会顺序的左移。如果package ID也被扩充到多于2bit，cluster 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域来的来确定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power-up或RESET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cluster ID、package ID以及processor 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cluster ID）可以表示成：

((Package ID << (1+((int)(log(2)(max(Logical_Per_Package-1,1)))) || Logical Processor 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方式（specific 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Processor 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（mask）和一个移位值（bit-shift value）
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

本节介绍了要想运行在IA-32超线程处理器上,操作系统要做哪些修改？同时也介绍了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 kernel 2.4.0针对超线程IA-32处理器所作的修改中具有代表性的。额外的针对超线程IA-32处理器的优化在Pentium 4 and Intel Xeon Processor Optimization Reference Manual中介绍（参考1.4 “相关文献”）。

7.7.6.1 在spin-wait loop中使用PAUSE指令

在Intel Xeon或Pentium 4处理器中，建议在所有的spin-wait loop中使用PAUSE指令。

使用spin-wait loop的软件包括多处理器同步原语（synchronization primitives）(spin-lock, 信号量, 互斥变量)和空闲循环（idle loop）。这种代码在等待某个资源的时候，使处理器一直执行load-compare-branch循环。在这个循环里包含一个PAUSE指令会大大提高效率（参考7.7.2 “PAUSE指令”）。下面的代码给出了一个spin-wait loop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:
```

上面的循环使用了"test, test-and-set"技术来确定同步变量的有效性。建议在书写spin-wait loop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 Idle loop中使用MONITOR/MWAIT

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OS上，典型的OS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 Halt 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 Halt instruction
VOID C1Handler()
{
```

```
    STI
    HLT
}
```

7.7.6.2 停止空闲逻辑处理器（HALT IDLE LOGICAL PROCESSORS）

如果两个逻辑处理器中的一个处于空闲状态或者在spin-wait loop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他逻辑处理器使用。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power-saving模式。

C1 Idle Loop中使用MONITOR/MWAIT

在C1 idle loop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
```

[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.

```
        // on Idle time accumulated
        IF (IdleTime >= IdleTimeThreshhold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1
            // handler shown below
        }
    }
}
// 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（guideline）

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（processor affinity）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 去掉基于执行的计时循环（EXECUTION-BASED TIMING LOOPS）

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(the local APIC timer or the timestamp counter））

进一步的信息，请参考*Pentium 4 and Intel Xeon Processor Optimization Reference Manual*（Section 1.4., “Related Literature”）

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、线程或其他代码区域的时候，Intel推荐在一个cache line中只放置一个锁或信号量。在Intel Xeon处理器MP中（有128字节的cache line），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 11 章　Intel® MMX™技术系统编程

本章描述在设计或增强操作系统对MMX技术的支持时，必须考虑的一些Intel® MMX™技术特点。它包括MMX指令集仿真、MMX状态(state)、MMX寄存器映射、MMX状态保存、任务和环境切换考虑、异常处理以及MMX代码调试。

11.1 MMX 指令集仿真

与x87 FPU指令不同，IA-32架构不支持MMX指令仿真。控制寄存器CR0中的EM标志(用于激活x87 FPU指令仿真)不能用于MMX指令仿真。如果设置了EM标志时执行MMX指令，将产生无效操作码异常(#UD)。表11-1给出了执行MMX指令时控制寄存器CR0中EM、MP和TS标志的相互影响。

Table 11-1. Action Taken By MMX Instructions for Different Combinations of EM, MP and TS

CR0 Flags			Action
EM	MP*	TS	
0	1	0	Execute.
0	1	1	#NM exception.
1	1	0	#UD exception.
1	1	1	#UD exception.

Note:

* For processors that support the MMX instructions, the MP flag should be set.

11.2 MMX 状态与 MMX 寄存器映射

MMX状态包括8个64位寄存器(MM0～MM7)。这些寄存器映射到浮点指针寄存器R0～R7的低64位(位0～63)(见图11-1)。值得注意的是，MMX寄存器是映射到浮点指针寄存器(R0～R7)的物理位置，而不是指针寄存器堆栈(ST0～ST7)中寄存器的相对位置。因此，MMX寄存器映射是固定的，并且不受浮点指针状态字中栈顶(TOS)域(位11～13)值的影响。

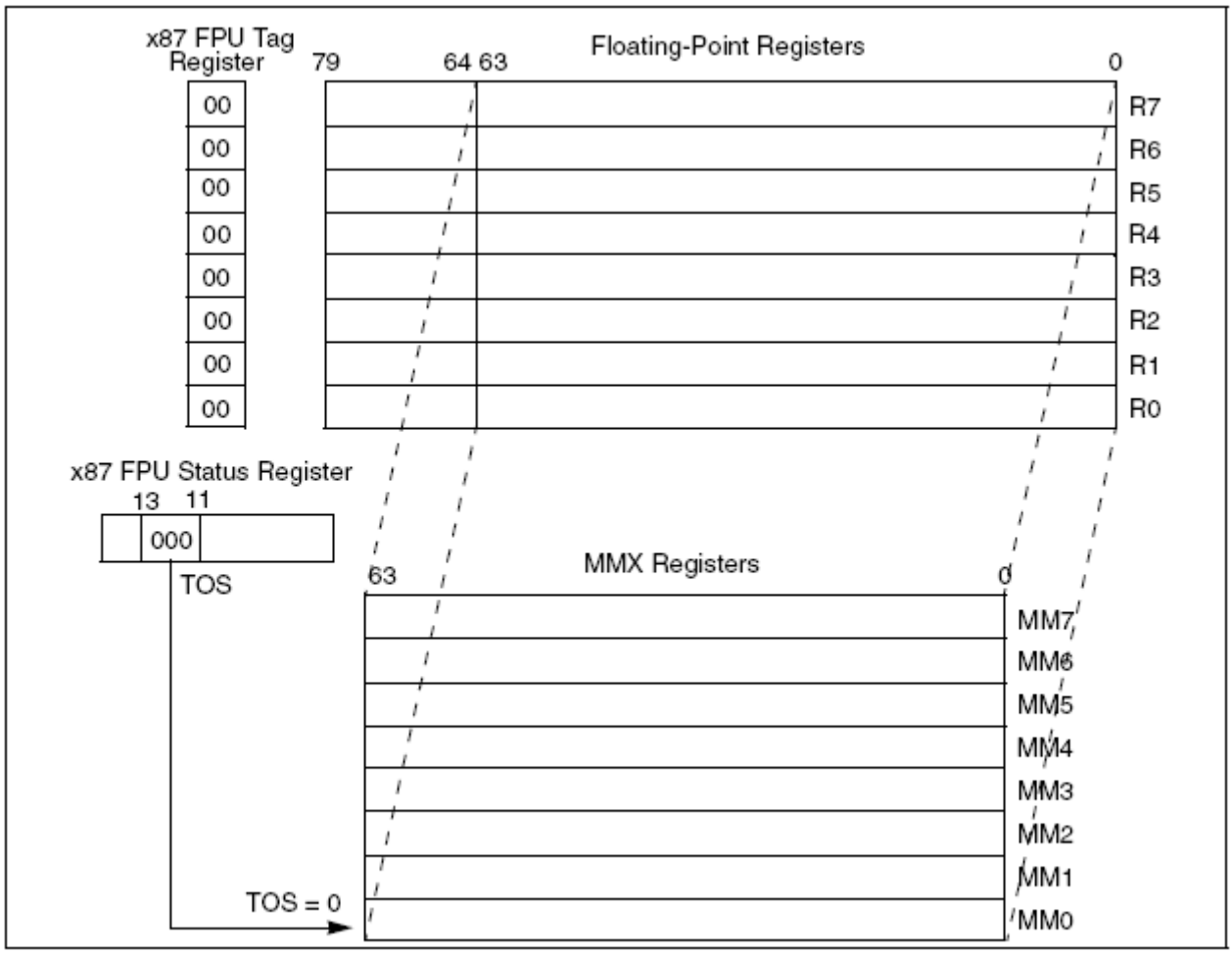

Figure 11-1. Mapping of MMX Registers to Floating-Point Registers

当用MMX指令往某个MMX寄存器中写入值时，这个值也出现在相应的浮点指针寄存器位0～63的位置上。同样的，用x87 FPU指令往某浮点指针寄存器写入浮点指针值时，其低64位也出现在相应的MMX寄存器中。

MMX指令执行时对浮点指针寄存器中的x87 FPU状态、x87 FPU标签字及x87 FPU状态字产生一些影响。这些影响如下：

- 在MMX指令往某MMX寄存器中写入值时，对应浮点指针寄存器的位64～79全部置1。

• 在执行某一MMX指令(EMMS指令除外)时，每个x87 FPU标签字中的标签域置为“00(有效)”。(也见11.2.1节，“MMX、x87 FPU、 FXSAVE及FXRSTOR指令对x87 FPU标签字的影响”)

• 在执行EMMS指令时，每个x87 FPU标签字中的标签域置为“11(空)”。

• 每次执行MMX指令时，TOS值置为“000”。

执行MMX指令并不影响x87 FPU状态字的其他位(位0～10, 14, 15)或组成x87 FPU状态(x87 FPU控制字、指令指针、数据指针或操作码寄存器)的其他x87 FPU寄存器内容。

表11-2归纳了MMX指令对x87 FPU状态的影响。

Table 11-2. Effects of MMX Instructions on x87 FPU State

MMX Instruction Type	x87 FPU Tag Word	TOS Field of x87 FPU Status Word	Other x87 FPU Registers	Bits 64 Through 79 of x87 FPU Data Registers	Bits 0 Through 63 of x87 FPU Data Registers
Read from MMX register	All tags set to 00B (Valid)	000B	Unchanged	Unchanged	Unchanged
Write to MMX register	All tags set to 00B (Valid)	000B	Unchanged	Set to all 1s	Overwritten with MMX data
EMMS	All fields set to 11B (Empty)	000B	Unchanged	Unchanged	Unchanged

11.2.1 MMX、x87 FPU、FXSAVE 及 FXRSTOR 指令对 x87 FPU 标签字的影响

表11-3归纳了MMX、x87 FPU、 FXSAVE及FXRSTOR指令对x87 FPU标签字中标签的影响，及对存在内存的标签字映像中相应标签的影响。

x87 FPU标签字域中的值并不影响MMX寄存器中的内容，也不影响MMX指令的执行。但是，正如11.2节“MMX状态与MMX寄存器映射”中所讲的，MMX指令会修改x87 FPU标签字的内容。如果在x87 FPU指令执行之前x87 FPU状态没有初始化或者恢复，这些修改将影响x87 FPU的操作。

注意，FSAVE、FXSAVE及FSTENV指令(这些用来存储x87 FPU状态信息)读x87 FPU标签寄存器及每个浮点指针寄存器的内容，决定每个寄存器的实际标签值(空、非零、零或专用)，并存储已更新的标签字到内存。执行完这些指令后，x87 FPU标签字中的所有标签

设为“空(11)”。同样的，EMMS指令通过设置x87 FPU标签字中的所有标签为“11”，来清除MMX/浮点指针寄存器中的MMX状态。

Table 11-3. Effect of the MMX, x87 FPU, and FXSAVE/FXRSTOR Instructions on the x87 FPU Tag Word

Instruction Type	Instruction	x87 FPU Tag Word	Image of x87 FPU Tag Word Stored in Memory
MMX	All (except EMMS)	All tags are set to 00B (valid).	Not affected.
MMX	EMMS	All tags are set to 11B (empty).	Not affected.
x87 FPU	All (except FSAVE, FSTENV, FRSTOR, FLDENV)	Tag for modified floating-point register is set to 00B or 11B.	Not affected.
x87 FPU and FXSAVE	FSAVE, FSTENV, FXSAVE	Tags and register values are read and interpreted; then all tags are set to 11B.	Tags are set according to the actual values in the floating-point registers; that is, empty registers are marked 11B and valid registers are marked 00B (nonzero), 01B (zero), or 10B (special).
x87 FPU and FXRSTOR	FRSTOR, FLDENV, FXRSTOR	All tags marked 11B in memory are set to 11B; all other tags are set according to the value in the corresponding floating-point register: 00B (nonzero), 01B (zero), or 10B (special).	Tags are read and interpreted, but not modified.

11.3 保存和恢复 MMX 状态和寄存器

因为MMX寄存器是映射到x87 FPU数据寄存器的，所以MMX状态可以按如下方法保存到内存或者从内存中恢复。

• 执行一条FSAVE、FNSAVE或FXSAVE指令来存储MMX状态到内存。(FXSAVE指令也存储XMM和MXCSR寄存器的状态)

• 执行一条FRSTOR或FXRSTOR指令来从内存中恢复MMX状态。(FXRSTOR指令也恢复XMM和MXCSR寄存器的状态)

上面所讲的保存和恢复方法操作系统的支持。(见11.4节“任务和内容切换时保存MMX状态”)有些时候应用程序按如下方法保存和恢复MMX寄存器。

• 执行八条MOVQ指令来保存MMX0～MMX7寄存器的内容到内存。然后执行EMMS指令(可选)来清除x87 FPU中的MMX状态。

• 执行八条MOVQ指令来从内存中读取先前保存的MMX寄存器内容到MMX0～MMX7寄存器。

注意：IA-32结构不支持搜寻x87 FPU标签字，然后只保存那些有效的条目。

11.4 任务和环境切换时保存 MMX 状态

从当前任务或环境切换到其它时，经常要保存MMX状态。一般来讲，如果现存的操作系统任务切换代码包括保存x87 FPU状态的工具，那么这些工具也依赖于保存MMX状态的代码而不用重写任务切换代码。因为MMX状态是映射到x87 FPU状态的(见11.2节“MMX状态和MMX寄存器映射”)，所以这一依赖是可能的。

随着FXSAVE及FXRSTOR指令和SSE/SSE2/SSE3的介绍扩展至IA-32结构，创建操作系统中的状态保存工具或者一个操作保存x87 FPU/MMX/SSE/SSE2/SSE3状态的执行指令成为可能。12.5节“设计操作系统工具来自动保存任务或内容切换时x87 FPU、MMX、SSE、SSE2和SSE3的状态”描述了如何设计这些工具。本节描述的方法适用于在需要时仅保存MMX和x87 FPU状态。

11.5 MMX 指令执行时可能发生的异常

MMX指令不会产生x87 FPU浮点指针异常，也不会影响EFLAGS寄存器中的处理器状态标志位或者x87 FPU状态字。MMX指令执行时可能产生如下异常：

• 访问内存时的异常：

— 缺堆栈段(#SS)。

— 一般性保护(#GP)。

— 缺页(#PF)。

— 对准检查(#AC) (如果设定对准检查)。

• 系统异常：

— 无效操作码(#UD)(MMX指令执行时如果控制寄存器CR0中的EM标志置为1，见11.1节“MMX指令集仿真”)

— 设备不可用(#NM)(在控制寄存器CR0中的TS置为1时执行MMX指令，见12.5.1节“用TS

标志控制x87 FPU、MMX、SSE、SSE2和SSE3状态的保存”)

- 浮点指针出错(#MF)(见11.5.1节“x87浮点指针异常未决时MMX指令的影响”)
- 以上异常发生时不完善的异常处理程序可能导致间接出现其它异常。

11.5.1 x87 浮点指针异常未决时 MMX 指令的影响

如果x87浮点指针异常还未解决时处理器遇到MMX指令，处理器在执行MMX指令前产生x87 FPU浮点指针出错(#MF)，以便x87 FPU浮点指针出错异常处理程序对浮点指针异常进行处理。在异常处理程序执行时，保持x87 FPU状态并让它(x87 FPU状态)对处理程序是可见的。异常处理程序返回后，执行MMX指令，由此改变x87 FPU状态(见11.2节“MMX状态和MMX寄存器映射”)。

11.6 调试 MMX 代码

执行MMX指令时，IA-32结构调试工具的操作方法与执行其它IA-32结构指令一样。

为了从内存FSAVE/FNSAVE或FXSAVE映像中正确解释MMX或者x87 FPU寄存器内容，调试程序需要考虑x87 FPU寄存器相对于TOS的逻辑位置与MMX寄存器的物理位置之间的关系。

在x87 FPU环境中，ST *n*指的是相对于TOS偏移位置为*n*的x87 FPU寄存器。而x87 FPU标签字中的标签与x87 FPU寄存器(R0～R7)的物理位置相关联。MMX寄存器总是指寄存器的物理位置(MM0～MM7已映射到R0～R7)。图11-2给出了这种关系。这其中，内环指的是x87 FPU和MMX寄存器的物理位置，外环指的是相对于当前TOS的x87 FPU相对位置。

当TOS为0时(图11-2中情形A)，ST0指向浮点指针堆栈的物理位置R0。MM0映射到ST0，MM1映射到ST1，等等。

当TOS为1时(图11-2中情形B)，ST0指向物理位置R2。MM0映射到ST6，MM1映射到ST7，MMX映射到ST0，等等。

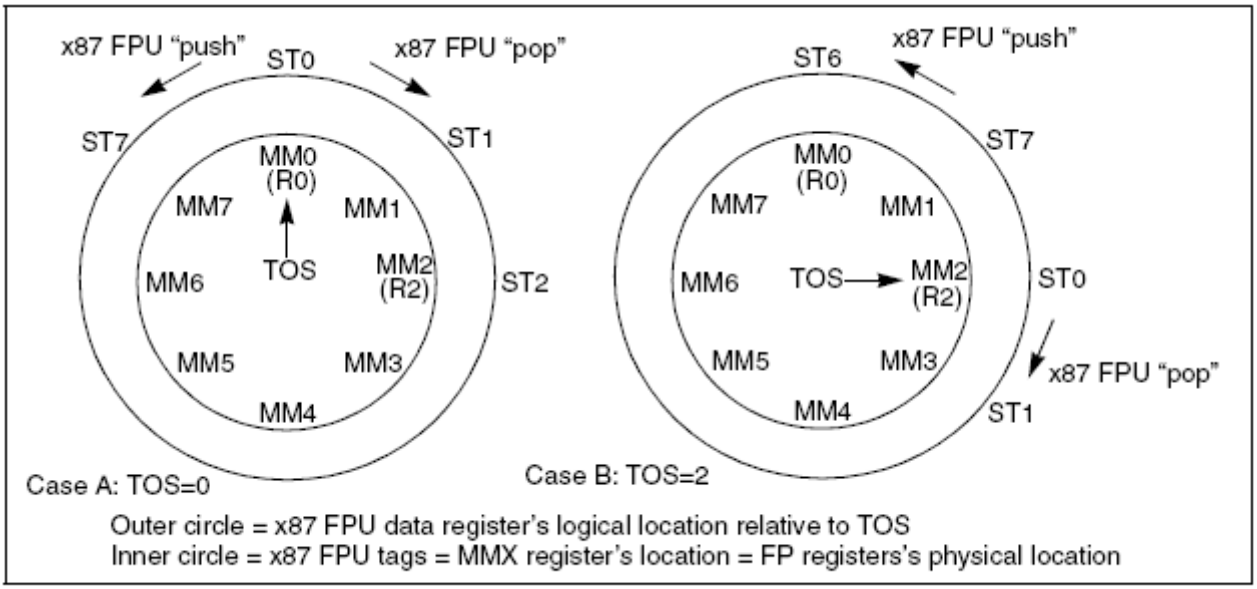

Figure 11-2. Mapping of MMX Registers to x87 FPU Data Register Stack

第 12 章 关于 SSE、SSE2 和 SSE3 方面的系统编程

本章描述了在设计或增强操作系统在对奔腾 3、奔腾 4 和至强处理器支持时必须要考虑的流式 SIMD 扩展（SSE)、流式 SIMD 扩展 2（SSE2）和流式 SIMD 扩展 3（SSE3）的特性。涉及的内容包括：打开 SSE/SSE2/SSE3 的扩展；为操作系统或执行程序提供 SSE/SSE2/SSE3 的支持；SIMD 浮点运算例外处理；其他情况的例外处理以及任务(上下文)切换时需考虑的因素。

12.1. 为操作系统提供 SSE/SSE2/SSE3 扩展的支持

为了使用 SSE/SSE2/SSE3 扩展，操作系统必须支持来初始化处理器，使之能使用流扩展功能；必须支持 FXSAVE 和 FXRSTOR 状态保存指令；必须能支持 SIMD 浮点例外处理。下面的各节给出了在操作系统或执行程序中提供这种支持的指南。由于 SSE/SSE2/SSE3 共享相同的状态、执行类似的操作，以下的指南对三套扩展同样适用。

IA-32 Intel系统架构 软件开发人员手册，第一卷的十一章 使用SSE2编程，十二章 使用SSE3 编程从应用程序的角度讨论了对SSE/SSE2/SSE3的支持。

12.5.1. 在操作系统中增加对SSE/SSE2/SSE3扩展的支持

下面的指南描述了操作系统中为了支持SSE/SSE2/SSE3扩展所必须进行的操作：

1．检查处理器是否支持SSE/SSE2/SSE3扩展。

2．检查处理器是否支持FXSAVE和FXRSTOR指令。

3．提供程序来初始化SSE/SSE2/SSE3的状态。

4．提供对FXSAVE、FXRSTOR指令的支持。

5．（如果必要的话）为SSE/SSE2产生的例外，提供非数字操作例外处理。

6．提供SIMD浮点运算例外处理。

下面各节描述了如何实现以上列举的各条指南。

12.5.2.　检查处理其是否支持SSE/SSE2/SSE3扩展

如果处理器试图执行它并不支持的SSE/SSE2/SSE3指令，会产生无效操作码（#UD）的例外。在操作系统试图使用SSE/SSE2/SSE3扩展前，它必须检查处理器是否支持这些指令。可以通过执行CPUID指令，并将其参数EAX设置为1，来进行检查。确保以下的比特位置位：

EDX：位25（表示处理器支持SSE）

EDX：位26（表示处理器支持SSE2）

EDX：位0（表示处理器支持SSE3）

12.5.3.　检查对FXSAVE、FXRSTOR指令的支持

另外，还需验证处理器对FXSAVE、FXRSTOR指令是否支持。

可以通过执行CPUID指令，并将其参数EAX设置为1，来进行检查。确保以下的比特置位：

EDX：24位(表示处理器支持FXSR指令)

12.5.4.　初始化SSE/SSE2/SSE3扩展

操作系统必须在应用程序使用SSE/SSE2/SSE3扩展前执行如下步骤进行初始化：

1．设置CR4的第9位（OSFXSR位）为1。设置这一位意味着操作系统会为使用FXSAVE、FXRSTOR指令来保存、恢复SSE/SSE2/SSE3的状态提供支持。这些指令在任务切换、触发SIMD浮点例外处理中被普遍使用来保存SSE/SSE2/SSE3的状态。（参见12.4 在任务（上下文）切换时保存SSE/SSE2/SSE3的状态，12.6 提供SIMD浮点例外处理程序）如果处理器不支持FXSAVE、FXRSTOR指令，在试图设置OSFXSR位时会产生#GR例外。

2．设置CR4的第10位（OSXMMEXCPT位）为1。设置这一位意味着操作系统会提供SIMD浮点例外（#XF）处理程序。（参见12.6 提供SIMD浮点例外处理程序）

CR4 中的 OSFXSR、OSXMMEXCPT 位必须由操作系统设置。处理器没有其它的方法来检测操作系统是否对 FXSAVE、FXRSTOR 指令以及对 SIMD 浮点处理例外的支持。

3．清除CR0的第2位（EM位）。本操作禁止模拟x87浮点处理器，这是使用SSE/SSE2/SSE3所必须的。（参见2.5 控制寄存器）

4．清除CR0的第1位（MP位）。这是所有IA-32处理器为了使用SSE/SSE2/SSE3指令所必须的。（参见9.2.1 配置x87浮点处理器环境）

表 12-1 显示了在下列条件下，处理器在执行第一条 SSE/SSE2/SSE3 指令时的行为：

- CR4中的OSFXSR、OSXMMEXCPT标志位
- CPUID指令返回的SSE/SSE2/SSE3的特征标志位
- CR0中的EM, MP和TS标志位

12-1.使用不同比特位时处理器的行为[1]

CR4		CPUID	CR0			行为
OSFXSR	OSXMMEXCPT	SSE，SSE2，SSE3	EM	MP[2]	TS	
0	X[3]	X	X	1	X	产生#UD例外
1	X	0	X	1	X	产生#UD例外
1	X	1	1	1	X	产生#UD例外
1	0	1	0	1	0	执行指令； 如果检测出未屏蔽的SIMD浮点例外则产生#UD例外
1	1	1	0	1	0	执行指令； 如果检测出未屏蔽SIMD浮点例外，则产生#XF例外
1	X	1	0	1	1	产生#NM例外

注：

1. 使用执行除PAUSE, PREFETCH h, SFENCE, LFENCE,MFENCE, MOVNTI, and CLFLUSH之外的其它SSE/SSE2/SSE3指令。
2. 对支持MMX指令的处理器，MP应置位。
3. X-不予考虑。

MXCXR寄存器的SIMD浮点例外屏蔽码（位7到12），刷新到0标志（位15），反向规格化为0标志（位6），舍入域（位13-14）必须保持其缺省值0，以便允许应用程序确定哪些特性被使用。

12.5.5.　为执行SSE/SSE2/SSE3指令时产生的例外提供处理程序

SSE/SSE2/SSE3和IA-32架构中的其它指令一样都能产生同样的内存访问例外(如：内存页故障、段不存在、地址超界)等非数值例外。

通常，已经存在的例外处理程序在不需进行修改时，能处理上面提到的和其它的例外。然而，针对现存的例外处理程序中使用的机制，可能需要进行必要的修改。

SSE/SSE2/SSE3能产生下列的非数值例外：

- 内存访问例外 ：
  - 不正确的操作码（#UD）。
  - 堆栈错（#SS）。
  - 一般保护错（#GP）。当访问未对齐的128位内存地址时，大部分SSE/SSE2/SSE3指令都能产生一般保护错。(MOVUPS/MOVUPD允许从/向128位未对齐的地址加载/存储数据，而不会产生一般保护错）在堆栈中访问未对齐16字节边界的128位地址时也会产生一般保护错，而不是堆栈错（#SS）。
  - 内存页错（#PF）。
  - 对齐检查（#AC）。当打开对齐检查时，只对小于128位的操作数：如16位、32位和64位的操作数进行检查。为了能打开对齐检查，需进行如下操作：
    - 设置AM标志(控制寄存器CR0位18)
    - 设置AC标志（EFLAGS的位18）
    - CPL必须设置为3

    如果打开对齐检查，MOVUPD/MOVUPS的16位/32位/64位操作数的不正确对齐会被检查出来。不保证能检查出128位操作数不正确对齐，并且这依赖于具体实现。
- 系统例外：
  - 不正确的操作码(#UD)。在下列条件下执行SSE/SSE2/SSE3指令时会产生这个例外：
    - CPUID指令返回的SSE/SSE2/SSE3特征标志为0，这个条件不影响CLFLUSH指令。
    - CPUID指令返回的CLFSH特征标志为0，这个例外仅在执行CLFLUSH指令时产生。
    - 无论TS标志是何值，CR0的EM标志（位2）为1。这种产生例外的条件对PAUSE, PREFETCHh, MOVNTI, SFENCE, LFENCE, MFENSE, 和CLFLUSH指令没有影响。
    - CR4寄存器的OSFXSR标志（位9）为0。该标志不影响这些指令：PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB,PMINSW, PMINUB, PMOVMSKB, PMULHUW,

PSADBW, PSHUFW,MASKMOVQ, MOVNTQ, MOVNTI, PAUSE, PREFETCHh, SFENCE,LFENCE, MFENCE和CLFLUSH。

- 当CR4的OSXMMEXCPT标志（位10）为0时执行一条指令触发了SIMD浮点例外处理程序。参见：“12.5.1使用TS标志来控制保存x87 FPU, MMX, SSE,SSE2和SSE3的状态”。

— 设备不可用（#NM）。这个例外会在CR0的TS标志（位3）设置为1时执行一条SSE/SSE2/SSE3指令产生。

其它的例外会由以上列举的例外间接产生。

12.5.6. 为SIMD指令提供浮点数例外（#XF）处理程序

SSE/SSE2/SSE3指令在对紧缩整数进行操作时不会产生数字例外。在对紧缩、标量单精度和双精度浮点数进行操作时会产生下列的（SIMD指令浮点）数字例外：

- 不正确的操作（#I）
- 被零除（#Z）
- 异常规格化的操作数（#D）
- 上溢（#O）
- 下溢（#U）
- 不精确的结果（精度有问题）（#P）

这些异常（除了异常规格化操作数例外）都定义在“IEEE 754 二进制浮点运算”的标准中。这些例外产生的条件和x87指令中触发x87产生浮点错误例外（#MF）的条件是相同的。

以上列举的例外都能够被屏蔽，这时处理器不会调用异常处理程序，而是向目标操作数返回合理的结果。然而，如果任何的例外没有被屏蔽，那么处理器在检测到例外后就会触发SIMD浮点数例外（#XF）。参见第五章 “中断19-浮点数例外（#XF）”。

为了处理没有被屏蔽的浮点数例外，操作系统必须提供例外处理程序。IA-32 Intel架构-软件开发人员手册-第一卷，第11章，SSE和SSE2浮点数例外描述了浮点数例外的种类，并给出了编写例外处理程序的建议。

为了表明操作系统提供了浮点数例外（#XF）处理程序，CR0寄存器的OSXMMEXCPT（位10）必须置位。

12.5.7. 数字错误标志和IGNNE#

SSE/SSE2/SSE3扩展忽略CR0的NE标志（总认为该标志位置位），并忽略IGNNE#管脚产生的

信号。当检测到未被屏蔽的SIMD浮点数例外后，总会产生一个SIMD浮点数例外（#XF）。

12.2. 模拟 SSE/SSE2/SSE3

IA-32架构不允许像x87指令那样允许模拟运行SSE/SSE2/SSE3指令。CR0寄存器的EM标志(用来请求模拟x87指令标志位）不能用来请求模拟SSE/SSE2/SSE3指令。如果在EM标志置位时执行SSE/SSE2/SSE3指令，那么就会产生不正确操作码（#UD）的例外。

12.3. 保存和恢复 SSE/SSE2/SSE3 的状态

SSE/SS2/SSE3的状态包括XMM和MXCSR寄存器的状态。建议采用下列方法保存、恢复该状态：

- 执行FXSAVE指令将XMM和MXCSR寄存器的状态保存到内存。
- 执行FXRSTOR从FXSAVE保存的内存中恢复XMM和MXCSR寄存器的状态。

这个保存、恢复方法对操作系统来说是必须的（参见12.5节，设计操作系统在上下文切换时自动保存x87、MMX、SSE、SS2和SSE3的状态）。

某些情况下，应用程序仅能使用下述方法保存MMX、MXCSR寄存器的状态：

- 执行8条MOVDQ将XMM0到XMM7的寄存器内容保存到内存中。
- 执行STMXCSR指令将MXCSR指令保存到内存中。

某些情况下，应用程序仅能使用下述方法恢复MMX、MXCSR寄存器的状态：

- 执行8条MOVDQ将XMM0到XMM7的寄存器内容从内存中读出。
- 执行LDMXCSR指令将MXCSR寄存器内容从内存中恢复。

12.4. 在任务或上下文切换时保存 SSE/SSE2/SSE3 的状态

当从一个任务切换到另一个任务，通常需要保存SSE/SSE2/SSE3的状态。FXSAVE、FXRSTOR提供了简单的方法来保存、恢复这个状态（参见12.3 节“保存、恢复SSE/SSE2/SSE3的状态”）这些指令也对保存x87、MMX状态也是有用的。关于如何编写该保存状态过程的指南参见12.5节，“设计操作系统来自动保存x87、MMX、SSE、SS2和SSE3的状态”。

12.5. 设计操作系统来自动保存 x87、MMX、SSE、SS2 和 SSE3 的状态

x87/MMX/SSE/SS2/SSE3的状态包括x87、MMX、XMM和MXCSR等寄存器的状态。FXSAVE、FXRSTOR指令提供了快速的方法保存、恢复该状态。如果操作系统中的任务/上下文切换部分已经实现，那么这段程序可以使用FSAVE/FNSAVE和FRSTOR来保存x87和MMX的状态，而且还能通过将FXSAVE、FXRSTOR替换FSAVE/FNSAVE、FRSTOR来保存、恢复SSE/SSE2/SSE3的状态。

如果任务/上下文切换必须从头编写，那么下面提供了一些方法能够通过使用FXAVE、FXRSTOR指令来保存、恢复x87/MMX/SSE/SSE2/SSE3的状态。

- 操作系统可以要求应用程序负责在任务挂起前保存x87、MMX、XMM和MXCSR寄存器的内容，并在任务重新开始运行前恢复这些寄存器。这个方法对协作型操作系统是合理的，因为这样的操作系统中应用程序控制/决定着何时任务发生切换，因此能在切换前保存这些寄存器的状态。
- 操作系统负责自动地在任务切换过程中保存x87、MMX、XMM和MXCSR寄存器的内容（使用FXSAVE指令），在挂起任务重新开始运行时自动恢复这些寄存器的内容（使用FXRSTOR指令）。这里，x87/MMX/SSE/SSE2/SSE3的状态必须作为任务状态的一部分，这个方法适用于抢先多任务系统，因为应用程序不知何时被抢占，因此不能在任务发生切换前有所准备，操作系统必须负责保存、恢复任务和x87/MMX/SSE/SSE2/SSE3的状态。
- 操作系统负责保存x87、MMX、XMM和MXCSR寄存器的内容，并将此作为任务切换的一部分，但延迟保存这些寄存器的状态，直到新任务执行与x87、MMX或SSE/SSE2/SSE3相关的指令。采用这个方法，仅当在新任务中执行与x87、MMX或SSE/SSE2/SSE3相关的指令时，才需要保存x87/MMX/XMM/MXCSR寄存器的内容（相关信息参见12.5.1 “使用TS标志来控制保存x87、MMX、SSE、SSE2和SSE3的状态”）。

12.5.1. 使用TS标志来控制保存x87、MMX、SSE、SSE2和SSE3的状态

使用FXSAVE保存x87/MMX/SSE/SSE2/SSE3的状态会使处理器有额外的开销。如果新任务不访问x87 FPU,MMX,XXM和MXCSR寄存器，那么可以通过在任务切换时不自动保存这些状态来避免给处理器带来的额外开销。

CR0寄存器的TS标志用来允许操作系统延迟保存x87 FPU/MMX/SSE/SSE2/SSE3的状态，直到新任务中实际执行了一条指令访问这些状态。如果TS标志置位，处理器就会监视指令流中

是否有x87 FPU/MMX/SSE/SSE2/SSE3指令。当处理器检测到其中任何一条指令，它会在执行指令前触发设备不存在（#NM）例外。那么设备不存在的例外处理程序就被用来保存前一个任务的x87 FPU/MMX/SSE/SSE2/SSE3状态（使用FXSAVE指令），并加载当前任务的x87 FPU/MMX/SSE/SSE2/SSE3的状态（使用FXRSTOR指令）。如果新任务从未执行x87 FPU/MMX/SSE/SSE2/SSE3相关指令，那么设备不存在的例外不会被触发，前一任务的状态也不会自动保留。

TS标志可以明确使用（通过对CR0执行MOV指令）或隐含使用（使用IA-32架构的任务切换机制）。当使用任务切换机制时，处理器在任务切换时自动设置TS标志。当设备不存在例外处理程序保存了x87 FPU/MMX/SSE/SSE2/SSE3的状态，处理器应使用CLTS指令清除TS标志。图12-1给出了使用TS标志保存x87 FPU/MMX/SSE/SSE2/SSE3状态的例子。在本例中，任务A是当前正在运行的任务，任务B是新任务。操作系统为每个任务维护了一个内存区域来保存x87 FPU/MMX/SSE/SSE2/SSE3状态并定义了变量（x87_MMX_SSE_SSE2_SSE3_StateOwner)来表示拥有该状态的任务。本例中，任务A是当前状态的属主。

在任务发生切换时，操作系统的任务切换代码必须根据当前x87 FPU/MMX/SSE/SSE2/SSE3状态的属主来执行以下的伪代码设置TS标志。如果新任务（本例中为任务B）不是当前状态的属主，TS标志置为1；否则，TS标志置为0。

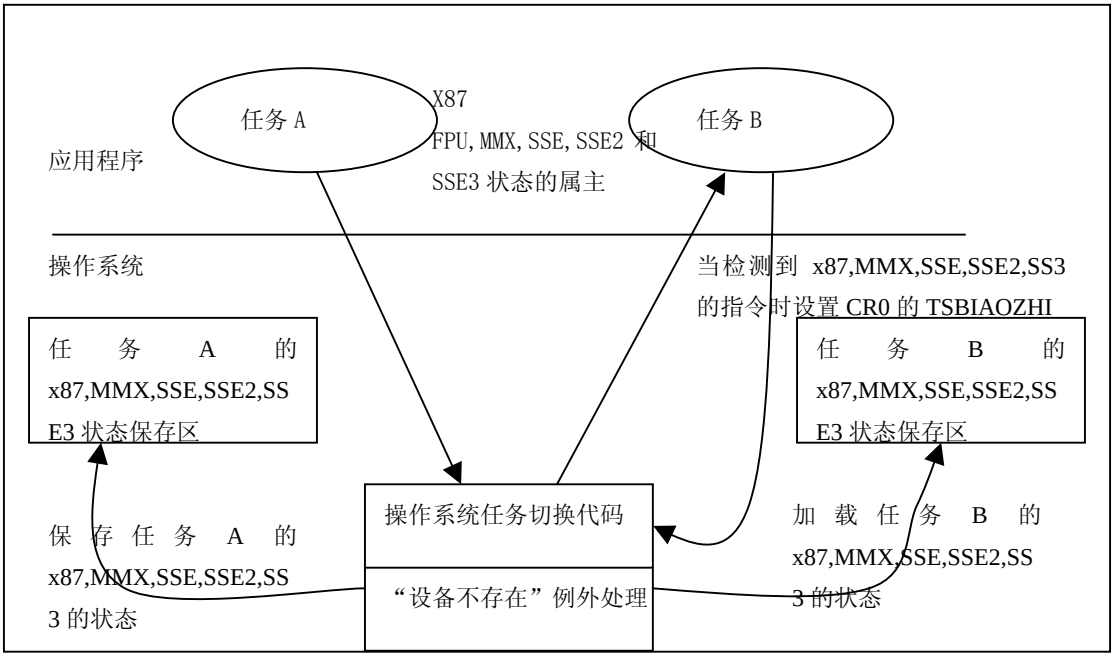

图 12-1.操作系统任务切换时保存 x87/MMX/SSE/SSE2/SSE3 状态

IF Task_Being_Switched_To ≠ x87FPU_MMX_SSE_SSE2_SSE3_StateOwner

THEN

```
CR0.TS ← 1;
ELSE
CR0.TS ← 0;
FI;
```

如果新任务在TS标志为1时试图访问x87 FPU、MMX、XMM或MXCSR，那么设备不存在例外（#NM）就会触发，该例外处理程序会执行如下伪代码：

```
FSAVE “To x87FPU/MMX/SSE/SSE2/SSE3 State Save Area for Current
x87FPU_MMX_SSE_SSE2_SSE3_StateOwner”;
FRSTOR “x87FPU/MMX/SSE/SSE2/SSE3 State From Current Task’s
x87FPU/MMX/SSE/SSE2/SSE3 State Save Area”;
x87FPU_MMX_SSE_SSE2_SSE3_StateOwner ← Current_Task;
CR0.TS ← 0;
```

这个例外处理执行如下任务：

- 替当前x87 FPU/MMX/SSE/SSE2/SSE3的属主将x87 FPU、MMX、XMM或MXCSR寄存器的内容保存到状态存储区。
- 从新任务的状态存储区恢复x87 FPU、MMX、XMM或MXCSR寄存器的内容
- 更新当前x87 FPU/MMX/SSE/SSE2/SSE3的属主为新任务
- 清除TS标志