段描述符缓冲

作者：Robert R. Collins

翻译：coly li

当你不知道一个东西是什么或者它是怎么工作的，你很可能会低估它的重要性。早在80286时期，所有的Intel x86处理器都包含一个叫做“段描述符缓冲”(segment-descriptor cache)的实体，它藏在后面，让我们感觉不到。每一个一个段寄存器被加载时他都会被更新。自从80286之后他被所有的Intel x86处理器的内存访问机制中所使用。如果你是一个最终用户，你很有可能使用到了以来于段描述符缓冲功能的软件程序；如果你是一个工程师，很有可能你就整天指望着它干活——但是从来没有认出它来。如果你是编写底层代码的工程师，对硬件编程，或者在保护模式下编程，那你可能需要了解段描述符缓冲，以及它是怎么工作的。

从80286到80486，段描述符缓冲的含义是含糊不定的，取决于保存段寄存器内部表现形式的内部处理器结构。这个表现形式包括段基址（segment base address），范围限制（limit），访问权限（access right）。在Pentium中，Intel为段描述符缓冲项引入了一个94项、2路的联合缓冲。因此，术语“段描述符缓冲”现在是不明确的，有两种可能的含义。更糟的是，新的断描述符缓冲被从Pentium Pro的设计中去掉了，但是在Pentium II中（缺少新的段描述符缓冲的Pentium Pro说明了它那可怜的16位性能）有引入回来了。这里，我将讨论自从80286就开始存在（并保持在所有现代的Intel x86处理器中）的原先的段描述符缓冲，和段描述符缓冲在微处理器内存管理中的角色。

加载描述符缓冲寄存器

无论在实模式、保护模式、虚拟8086模式或者系统管理模式下，微处理都将每一个段的基地址保存在一些隐藏的描述符缓冲寄存器中。每一个一个段寄存器被加载时，这个段的基地址，段大小限制和段的访问属性（访问权限）都被加载（缓冲）到这些隐藏的寄存器中。为了提高性能，之后的内存引用的访问都是通过访问这些描述符缓冲寄存器来实现的（而不是再次从内存中加载段寄存器）。如果没有这种优化，每一次内存访问都需要微处理器处理很多很消耗时间的任务。在实模式下，微处理器需要根据段寄存器的数值计算物理内存地址。访问权限总是同一个读/写数据段（甚至于代码段）关联起来。而limit将差不多总是64KB的内存。在保护模式下，段必须在适当的描述符表中进行查找。段基址是由描述符表的各个域联合起来构成的。段访问权限和段大小限制也是包含在描述符表中。对于每一次内存访问，微处理器总是需要访问这些数据结构。而这些信息都保存在内存中，因此相比较于微处理的速度而言，访问这些数据的速度实在是太慢了。如果没有一个内部的描述符缓冲将这些数值缓冲起来的话，每一次内存访问将会暗中需要很多其他的内存访问。

现在考虑到实模式和保护模式下的不同。如果段描述符缓冲不存在，测定段基址、大小限制、访问权限将需要多于一个CPU周期去完成。因此，段描述符缓冲的存在就消除了这些根本的差异。他的存在允许当每一个段被加载的时候，这些差异都可以被解决。对于性能的损失只会发生一次，内存管理中总是使用保存在段描述符缓冲中的对应于各自的段寄存器的数值。当上电时，描述符缓冲寄存器们按照缺省的固定的数值被加载——CPU处于实模式下，所有的段都数据段一样被标志为可读写，包括代码段。如果按照Intel所说的，每一个在实模式下加载段寄存器时，基地址通过段的值的16倍得到，而访问权限和大小限制都是由固定的“实模式兼容”数值给定的。但事实不是这样的。实际上，对于286、386和486， 每次段寄存器被加载时，只有CS描述符缓冲是按照固定值加载的。对于之后的Intel处理器，在

实模式下无论是加载 CS 或者其他任何的段寄存器，都不会改变访问权限和大小限制。从保存在描述符缓冲寄存器们中的先前设置的属性会被继承下来。因此我们甚至可以在 80386 的实模式下获得一个 4GB 的只读数据段——但是 Intel 从不承认这种操作模式，虽然暗中支持它。进一步，考虑到不让已有的利用到这点的程序失效，Intel 也不会将这个操作模式去除掉。

在以下方面，保护模式同实模式是不同的：当每一次某一个段寄存器被加载时，描述符缓冲寄存器都会全部被加载，没有任何数据会从前次设置中继承下来。描述符缓冲是直接从描述符表中加载的。CPU 通过测试保存在描述符表中的访问权限来检验段的合法性，非法的数值将会产生一个异常。任何尝试使用一个可读/写的数据段来加载 CS 的尝试都会产生一个保护错误，然后将数据段寄存器作为一个可执行段来加载的尝试也将产生一个异常。CPU 严格的执行这些保护规则。如果描述符表项通过了所有的测试，然后描述符缓冲寄存器才会被加载。

描述符缓冲寄存器的格式

每一代处理器的段描述符缓冲寄存器的布局结构几乎都会变化，但是他们的功能是不变的。这些差别是“实现相关”的差别，因为他们的具体布局和内容是依赖于微处理器整体的设计和实现的。多数情况下，段描述符缓冲中的域是镜像了保护模式的描述符表中的域。对于 32 位的描述符表表项，段基地址，段访问权限和段大小限制域都是不连续的，因此在放入段描述符缓冲之时会将这些相关信息先进行合并。图 1 展示了段描述符缓冲和描述符表项中各域之间的关系。

图 1：将描述符表中的域合并后放入段描述符缓冲

Offset	63..56	55	54	53	52	51..48	47	46..45	44	43..40	39..16	15..00

Description	Base[31:24]	G	D/B	0	AVL	Limit[19:16]	P	DPL	S	Type	Base[23:00]	Limit[15:00]

颜色说明

段基址　　段访问权限　　段大小限制

段描述符缓冲

对于了解段描述符缓冲中的结构布局是非常有用的。由于段基址、段大小限制等都是从描述符表中取出合并后建立一个段描述符的基地址和段大小限制的。段描述符缓冲中的访问权限的格式依据实现方法而各有不同。同样，缓冲中这些域的顺序也是变化的。但是知道段描述符缓冲的格式可以提高你的工作效率，减少开发和调试的时间。表 1—表 4 展示了 Intel x86 从 80286 到 Pentium Pro 的描述符缓冲项的格式。

Table 1 - 80286 Descriptor Cache Entry

Bit Position	47..32	31	30..29	28	27..24	23..00
Description	16-bit Limit	P	DPL	S	Type	24-bit Base

表 2 80386 和 80486 描述符缓冲项

Bit Position	95..64	63..32	31..24	23	22..21	20	19..16	15	14	13..0
Description	32-bit Limit	32-bit Base	0	P	DPL	S	Type	0	D / B	0

表 3 Pentium 描述符缓冲项

Bit Position	95..79	78	77..72	71	70..69	68	67..64	63..32	31..00
Description	0	D/B	0	P	DPL	S	Type	32-bit Base	32-bit Limit

表 4 Pentium Pro 描述符缓冲项

Bit Position	95..64	63..32	31	30	29..24	23	22..21	20	19..16	15..00

Description	32-bit Base	32-bit Limit	0	D/B	0	P	DPL	S	Type	Segment Selector

现实生活中的描述符缓冲寄存器

有很多方法来利用段描述符缓冲寄存器们。在系统管理模式（SMM）下你可以直接控制段描述符缓冲中的每一个域（参看 *DDJ* January/March/May 1997 的“an in-depth look at System Management Mode”专栏）。In-circuit emulators（ICEs）同样可以允许对段描述符缓冲的每一个域进行直接访问（参看 *DDJ* July/September/November 1997 “information on in-circuit emulation”专栏）。

例如，当编写任何低级汇编语言的程序时（例如OS Kernel，设备驱动，BIOS或者保护模式编程），我犯了一个普遍的简单的错误。有时候我在建立自己的段描述符表的时候会犯一个错误，通常是全局描述符表（GDT）。我可能会使用错误的段基址、段大小限制和访问权限建立一个GDT。然后我的程序就崩溃了，因此我不得不使用ICE来debug。我在我的代码中插入了未公开的ICEBP指令，好让ICE在发生错误的地方设置断点（参看 http://www.x86.org/secrets/opcodes/ICEBP.html）。没过多久，我就发现我在建立描述符表的时候使用了错误的数值。使用ICE，我可以加载段描述符缓冲中的每一个域。如果我使用了一个不正确的段基地址，我可以修正它然后继续运行。同样我可以对段大小限制和段访问权限域进行同样的修正。我知道这些数值都是固定的，也就是说在新的段寄存器值被加载之前，这些数值是不会改变的。因此，我可以进行这些改变，并且继续调试我的程序。使用这个技术，我通常可以在编译之前发现4到6个Bug。由于我不需要在发现和确认每一个问题后，不需要重新编译我的程序，因此节省了不少时间。

在SMM模式下编程就暗中使用了段描述符缓冲寄存器们的优势。段描述符缓冲寄存器们连同

微处理器的状态一起在 SMM 状态保留映象（SMM stat save map）中被保存和恢复。这些数值根据进入和退出系统管理模式（SMM）相应的被保存和恢复。在作者 1997 年 3 月的 DDJ 专栏中(March 1997 *DDJ* column),作者表述了 Pentium SMM 状态保留映射的所有没有公开的域信息（在 Intel 的谈话中被称为 reserved 的那些）。正如作者讨论的那样，段描述符缓冲寄存器们保存在这些被预留的域中。通过 SMM 处理程序来操作这些段描述符缓冲的数值是可能的。段基址可以被改变成一个与他对应的段寄存器不一致的数值。段访问权限可以从当前的 CPL-3（current-privilege-level 3）改为 CPL-0（显然违背了系统管理安全策略）。段大小限制可以被改变成建立一个实模式下的 4GB 的段。使用 SMM（系统管理模式），使得通过编程来改变段的属性值成为可能；例如，在 2MB 处的一个实模式的段，一个大小限制为 4BG-16 字节的段，或者一个在保护模式下的可读/写的代码段（不是那种在 CPL-3 的任务中进行 CPL-0 级访问的意思）。

描述符缓冲异常和建立 Unreal 模式

使用这些方式来操作段描述符缓冲是很有挑战性的。但是还有别的编程的办法来让段描述符缓冲工作——建立一个被称为 Unreal 模式的 CPU 操作模式。

Unreal 模式就是一个拥有 4GB 段大小限制时的实模式。Unreal 模式可以不依靠任何的硬件调试器或者用汇编语言编写的 SMM 程序。想象一下一个程序从实模式下开始运行，然后进入保护模式。一旦进入保护模式，程序会用设为 4GB 段大小限制的描述符加载所有的段寄存器们。当设置了段大小限制之后，立刻返回到实模式下，并且不将段寄存器们恢复为 64KB 的段（实模式兼容段）。一旦返回实模式后，段大小限制将会保持他们的 4GB 的限制。因此，DOS 程序可以不进入保护模式就利用整个的 32 位地址空间。

Unreal 模式自从在 80386 上被发现后，就得到了广泛的使用。Unreal 模式使用的太普及了，

以至于 Intel 不得不在之后的处理器中继续支持这种行为，虽然 Intel 从未公开声明这个特性。内存管理器和游戏经常利用 Unreal 模式。使用 Unreal 模式的 demo 源代码可以从 DDJ 的 Resource Center 获得，或者是从 ftp://ftp.x86.org/dloads/UNREAL.ZIP 下载。

实模式下的代码段（CS）描述符缓冲的行为在每一代的 Intel 处理器中都不尽相同。代码段描述符缓冲在实模式下的角色在 80286，80386 和 80486 以及所有之后的 Intel 微处理器中都是有所不同的：早期微处理会在实模式下继承保护模式下的访问权限直到一个段外的控制跳转发生；之后的处理器不管有没有段外（far）控制转移都忽略了 CS 描述符缓冲中的访问权限。早期的处理器，任何的段外控制转移都会将 CS 描述符缓冲的访问权限设置为它的实模式兼容值——实模式下的可读/写数据段(value=0x93)。之后的处理器会保持原先的数值不变，但是会忽略掉他的内容。因此，在后期处理器上从实模式切换到保护模式会立刻将 CS 描述符缓冲中的访问权限设置为正确的值（感觉这里是不是写反了？）。在早期的处理器上，CS 待续奥限制也被恢复为他的实模式兼容值（64KB）。后期的处理器没有考虑 CS 段大小限制，使得它的行为同其它的数据段寄存器相一致。

从 80286 到 Pentium，所有的 Intel 处理器的当前优先级（CPL）都是起源于 SS（堆栈段）的访问权限。当 SS 寄存器被加载时，CPL 会根据 SS 描述符表项而被加载。未公开的 LOADALL 指令（或者系统管理模式 RSM 指令）可以用来操作 SS 描述符的访问权限，这样就可以实现对微处理器的 CPL 的直接修改。（参看 http://www.x86.org/articles/loadall/描述关于 LOADALL 指令的描述）。Pentium Pro 的行为有所不同：一 CPL 被加载到 Pentium Pro，他并不是内在的起源于 SS 的访问权限。Pentium Pro 中有一个单独的 CPL 寄存器。通过系统管理模式的 RSM 指令，你可以直接操作 Pentium Pro 的 CPL 寄存器，而不是通过操作 SS 的访问权限数值（我会在未来的专栏中介绍 Pentium Pro SMM 状态保留映射和它的所有秘密）。

小结

我每天都在使用段描述符缓冲寄存器们——无论我是在在我的 ICE 上来修改通常的保护模式编程错误，还是在系统管理模式下创建事件，或者创建一个可以寻址整个 4GB 地址空间的实模式段。段描述符缓冲的用法同处理器的实现是密切相关的，也就是说，对于段描述符缓冲的行为和解构是取决于特定的微处理器的。Intel 没有承诺在不同型号的微处理器之间会保持相同的描述符缓冲的行为。因此，编写依靠这种行为的产品质量级的代码的行为，最终都会被认为是有勇无谋（Unreal 模式除外）。