x86 保护机制

孔俊

2022-10-12

Contents

1 保护机制的开启与关闭 **2**

2 描述符表限长检查 **2**

3 段限长检查 **2**

4 段类型检查 **3**
- 4.1 类型信息的存储 . . . 3
- 4.2 类型检查 . . . 3
- 4.3 空选择子的检查 . . . 3

5 特权级检查 **4**
- 5.1 访问数据段时的特权级检查 . . . 5
- 5.2 访问代码段中的数据 . . . 6
- 5.3 堆栈寄存器 SS 的特权级检查 . . . 6
- 5.4 在不同代码段之间进行程序控制转移时的特权级检查 . . . 6
- 5.5 直接调用或跳转到代码段（Dirct Calls or Jumps to Code Segments） . . . 6
  - 5.5.1 访问非一致代码段 . . . 7
  - 5.5.2 访问一致代码段 . . . 8
- 5.6 通过调用门访问代码段 . . . 9
  - 5.6.1 IA-32 架构下的调用门 . . . 9
  - 5.6.2 64 位处理器 (IA-32e mode) 下的调用门 . . . 10
  - 5.6.3 通过调用门访问代码段的过程 . . . 10
  - 5.6.4 堆栈切换 . . . 12
    - 5.6.4.1 IA-32 下的堆栈切换 . . . 13
    - 5.6.4.2 IA-32e 模式下的堆栈切换 . . . 14
    - 5.6.4.3 从被调用过程返回 . . . 14

1 保护机制的开启与关闭

在 x86 体系结构中，段的保护机制在 CPU 进入保护模式是自动开启，没有相应的关闭机制; 页的保护机制在开启分页内存管理后自动开启，没有相应的关闭机制。如果需要关闭段、页的保护机制，可以通过将段、页的访问特权降到最低实现。本文不涉及页机制下的保护机制。

2 描述符表限长检查

CPU 使用选择子获取段描述符，然后再使用段描述符访问内存。在使用选择子访问描述符时，必须保证选择子指向的段描述符在相应的描述符表中，因此要进行选择子和描述符表限长检查。

- 全局结构：GDT 和 IDT

在 GDTR 和 IDTR 中有 16 位存放表限长，在使用选择子访问 GDT 或 IDT 时将选择子中记录的描述符位置（描述符表中的下标）和表限长比较即可。

- 局部结构：**LDT** 和 **TSS**

  在 LDTR 和 TR 的不可见部分（shadow part) 存储着对应 LDT 和 TSS 的表限长，在访问 LDT 或 TSS 时选择子描述符位置（描述符表中的下标）和表限长比较。

在 IA-32e 模式下仍然会进行描述符表限长的检查。

3 段限长检查

在获取到段描述符后生成线性地址时，还必须保证线性地址在被访问的段内，所以必须进行偏移地址和段限长的检查。

对于非向下拓展（expand down) 的段，段限长就代表了程序能够访问的最高有效地址。如果段中 G flag 为 0（段限长以字节为单位，段最长为 1MB），那个 20 比特的段限长就是段中的最大偏移，所有的偏移都应该小于等于段限长; 如果段中 G flag 为 1（段限长以 4KB 为单位，段最长为 4G），还要注意在计算时段限长会被拓展（scalling)，类似与 C 语言中数组元素地址的计算。比如，G flag 为 1, 段限长为 0, 此时段的最大长度为 4KB，偏移必须在 [0,4KB-1] 中。

对于向下拓展的数据段是专门为堆栈准备的，因为堆栈从高地址到低地址增长，段限长有着不同的机制，但在 intel 手册 • [1]中我没有读到详细的介绍，网上我只找到了How to Use Expand Down Segments on Intel 386 and Later CPUs，但是没有读懂。如果有人读懂了，欢迎指教。

在 IA-32e 模式下，32 位兼容模式会进行段限长的检查，64 位模式不进行段限长的检查（毕竟根本就不使用段）。

[1] 本文说指的 intel 手册是*Intel® 64 and IA-32 Architectures Software Developer’ s Manual Volume 3 (3A, 3B, 3C & 3D):System Programming Guide*

4 段类型检查

在对段和段选择子进行操作时，还可能会发生对段的类型信息的检查。这里的类型信息不仅仅指段的类别还包括读写等权限。

4.1 类型信息的存储

在段描述符中类型信息存储在两个地方

- **S** 标志位：为 0 表示描述符指向的段是系统段（TSS、LDT），为 1 表示描述符指向的段是应用段（数据段、代码段）。
- **TYPE** 域：设置段的特性，比如读写权限。

4.2 类型检查

对段的类型的检查在以下情形中发生（Intel 手册并没有完全列举）：

- 当选择子被加载到段寄存器中时，特定的寄存器只能存放指向特定类型的段描述符的选择子
  1. CS 寄存器只能存放代码段的选择子;
  2. 系统段和不可读的代码段的选择子不能存放在数据段寄存器（DS,ES，GS 等）中;
  3. 只有可读的数据段选择子可以存放在 SS 中。
- 当段选择子被加载到 LDTR 或者 TR 中时：LDTR 只能存放指向 LDT 的选择子，TR 只能存放指向 TSS 的选择子。
- 当指令访问相应的段时：指令不能向代码段写入数据，不能向只读的数据段写数据。
- 当指令以选择子作为运算对象时：选择子必须和指令匹配，比如 far JMP 不应该以代码段选择子为对象。
- 某些内部操作中：比如，通过 TSS 执行任务切换时，会检查 TSS 指向的对应的段。

4.3 空选择子的检查

有时选择子可能是空的（相当与 C 语言中的 NULL），在加载空选择子到 CS 或 SS 中会产生 #GP 异常，加载到 DS、ES、FS、GS 中不产生异常，但在访问时会导致 #GP 异常。

在 IA-32e 的 64 位模式中不使用段寄存器选择描述符，因此不进行空段选择子的检查。

5 特权级检查

在 X86 中，处理器有 4 个特权级（level 0 到 level 3），特权级数值越小级别越高。通常，系统仅使用两个特权级，操作系统处于 level 0, 应用程序处于 level 3。从程序的角度看这就是内核态和用户态。

由于特权级的数值和级别恰好相反，为了避免混淆，在本文中使用高低一词时特指级别，使用大小一词时特指数值。

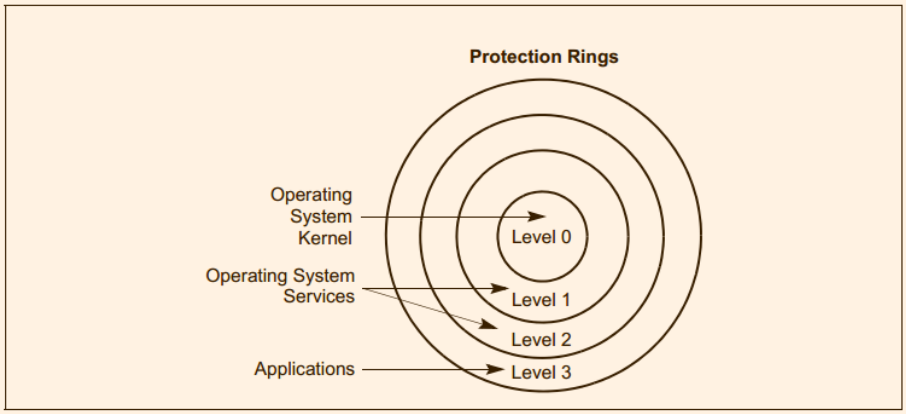

Figure 5-3. Protection Rings

在通过段或门访问内存时，会比较当前程序/任务的特权级和要访问的内存中的代码/数据特权级（可以类比 UNIX 的用户权限）。

X86 中的特权级实际上指的是 CPL(Current privilege level)、DPL(descriptor privilege level) 和 RPL(requested privilege level)。

CPL 代表当前执行的程序/任务的特权级。CPL 存储在 CS[2]中（第 0、1 位）。通常 CPL 等于正在执行的指令所在的代码段的特权级。当程序的控制传送到不同特权级的非一致 (non-conforming) 代码段时，CPL 也会被相应地修改。但是当控制传送到一致 (conforming) 代码段时，CPL 不会修改。

DPL 代表段或门的特权级，存储在相应的描述符中。当正在执行的程序试图访问段或门时，DPL 就会被拿来和 CPL 和 RPL 比较。对于不同的门和段，DPL 有着不同的含义：

- 数据段：DPL 代表能够访问该段的最低特权级。比如, 某个数据段的特权级是 1, 那么就只有在特权级 0/1 的程序可以访问这个段。
- 非一致代码段（不通过调用门）：DPL 代表能够访问该段的程序所处的特权级。比如，某个非一致代码段在特权级 0, 就只有特权级 0 的程序可以访问它。
- 一致代码段（不通过调用门）：DPL 代表能够访问该段的最高特权级。
- 调用门：DPL 代表能够访问该门的最低特权级。
- 一致或不一致代码段（通过调用门访问）：DPL 代表能够访问该代码段的最高特权级。比如，某个一致代码段在特权级 2，在特权级 0 或 1 的程序不能访问该段。

[2] intel 手册 5.5 节里面说 CPL 存储在 CS 和 SS 中，是因为 SS 的 DPL 必须和 CPL、DPL 相同。从 intel 手册（5.6,5.7）的语境看，CPL 应该特指 CS 中的特权级（0、1 位）。

- **TSS**：DPL 代表能够访问该 TSS 的最低特权级。

RPL 存储在其他段寄存器的低 0、1 位，它和 CPL 搭配使用。处理器会检查 CPL 和 RPL 的值，来决定程序能否访问某个段。RPL 的作用会在下文逐步介绍。

特权级检查发生在段选择子被加载到段寄存器中时。对于数据段的检查与程序控制转移涉及到的数据段不同，所以分开介绍。

5.1 访问数据段时的特权级检查

段选择子被加载到段寄存器时，处理器通过比较 CPL（当前执行的程序的特权级）、RPL（段选择子的特权级）和 DPL（数据段的特权级）来判断程序能否访问该段。CPL 和 RPL 必须小于或等于对应的 DPL，否则无法访问数据段，产生 #GP 异常。

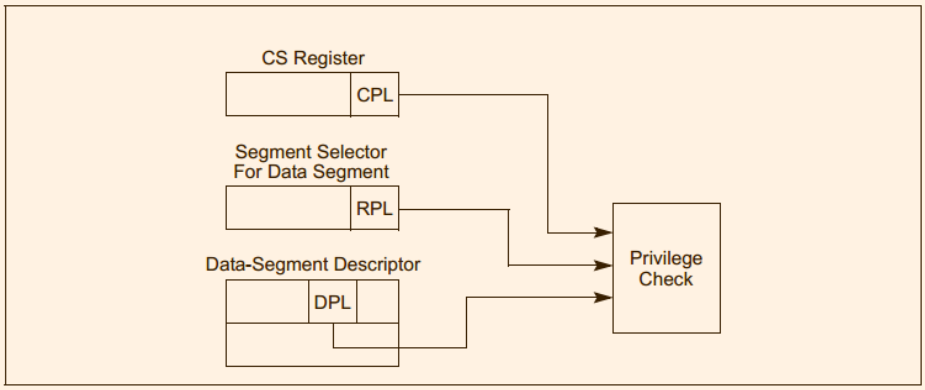

Figure 5-4. Privilege Check for Data Access

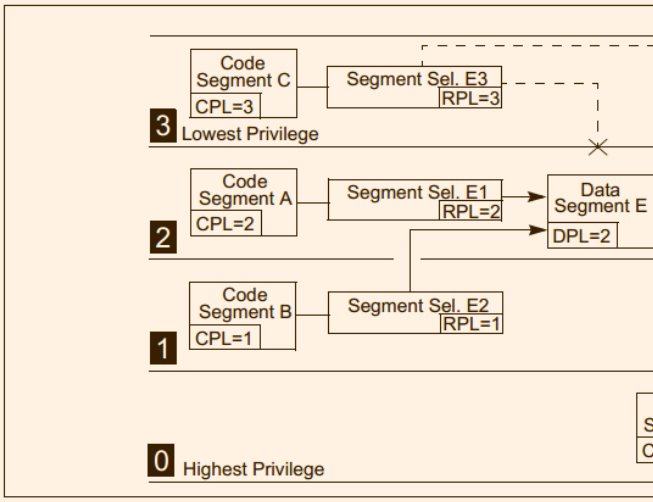

Figure 5-5. Examples of Accessing Data Segments From Va

这几个例子详细说明了访问数据段时的检查过程。

- 代码段 C 的 CPL 和选择子 E 的 RPL 大于数据段 E，所以代码段 C 中的程序不可以访问数据段 E。

- 代码段 A、B 的 CPL、RPL 均小于数据段 E，所以代码段 A、B 中的程序可以访问数据段 E。
- 代码段 D 的 CPL 小于数据段 E，但 RPL 大于数据段 E，所以代码段 D 即使特权级更高也无法访问数据段 E。

由于数据段寄存器是可以直接被用户修改的，RPL 和 CPL 的“重写”机制避免了处于低特权级的程序访问高特权级的数据段。

5.2 访问代码段中的数据

有些情况下需要访问代码段中的数据结构，X86 有以下几种可能的方法：

1. 将非一致可读代码段的选择子加载到数据段寄存器中。
2. 将一致可读代码段的选择子加载到数据段寄存器中。
3. 使用代码段前缀（*code-segment override prefix)*CS 来读取可读代码段。

三种方法都进行上文介绍的特权级检查。第一种方法可能无效（CPL 为 3,DPL 为 0,RPL 为 0, 最终无法访问），第二、三种方法肯定成功。

5.3 堆栈寄存器 **SS** 的特权级检查

堆栈段是特殊的数据段，SS 中的选择子特权级（RPL）必须同时和 CPL、DPL 相等。

5.4 在不同代码段之间进行程序控制转移时的特权级检查

X86 可以通过 JMP、CALL、RET、SYSENTER、SYSEXIT、SYSCALL、SYSRET、INT n 和 IRET 指令进行控制的转移。X86 多样的控制转移机制（有些还涉及到中断处理机制）增加了特权级检查的复杂度。这篇文章不介绍涉及中断的处理的控制转移机制。

5.5 直接调用或跳转到代码段（**Dirct Calls or Jumps to Code Segments**）

近转移形式的 JMP、CALL、RET 指令仅将程序的控制从现在的执行点传送到当前代码段的另一个执行点，因为不直接其他的代码段，所以不需要进行特权级检查。

当将程序的控制传送到另一个代码段（不通过调用门）时，处理器会检查以下四种信息：

- CPL：当前执行程序的特权级
- DPL：目标代码段的特权级
- RPL：目标代码段选择子的特权级
- C flag：目标代码段是非一致（C flag 为 0）的还是一致（C flag 为 1）的

处理器使用 CPL、RPL 和 DPL 判断能否访问代码段的规则取决于 C flag 的值，也就是说对于一致代码段和非一致代码段有不同的规则。

RPL 在目标代码段的选择子中，这个选择子是 JMP、CALL 的运算对象而不是 CS 中的值。

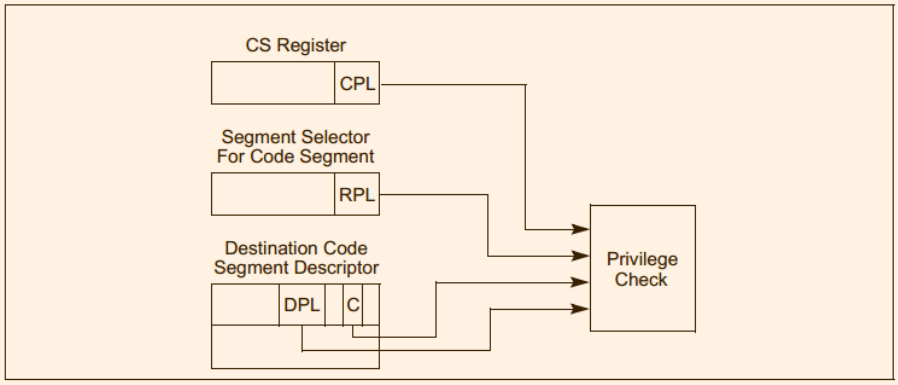

Figure 5-6. Privilege Check for Control Transfer Without Using a Gate

5.5.1 访问非一致代码段

在访问非一致代码段时，CPL 必须等于 DPL，并且 RPL 必须小于等于 CPL，否则产生 #GP 异常。当选择子被加载到 CS 中时，不管 RPL 是多少都不会修改 **CPL**。

下图中的例子详细说明了控制转移到一致代码段时的特权级检查。

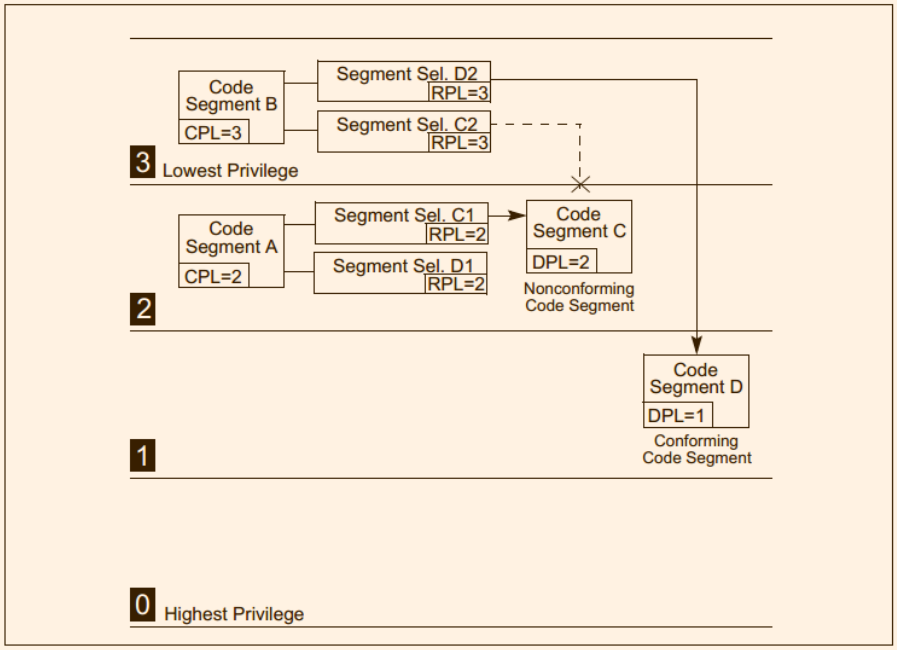

Figure 5-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

- 代码段 C 是非一致的，因为 A 代码段 CPL 是 2, 等于 C1 和 D1 的 RPL，并且等于代码段 C 的 DPL，所以 A 代码段中的过程可以调用 C 中的过程。
- 因为代码段 B 的 CPL 不等于代码段 C 的 DPL，所以代码段 B 中的过程不能调用代码段 C 中的过程。

RPL 在其中起的作用非常有限，主要还是看 CPL 和 DPL。因为 RPL 只需要小于等于 CPL 就可以了，所以 C1 设置为 0、1 都可以让代码段 A 中的过程成功调用代码段 C 中的过程。因为控制转移到非一致代码段时 CPL 不改变，所以不能够试图通过选择子修改 CPL。

5.5.2 访问一致代码段

在访问一致代码段时，调用过程的 CPL 必须大于等于目标代码段的 DPL，否则产生 #GP 异常，RPL 不被检查。

一致代码段中的 DPL 代表能够访问该代码段的程序的最高特权级，只要特权级低于一致代码段就可以成功访问。

在上图中，代码段 A、B 的特权级都低于代码段 D，所以 A、B 中的程序可以访问 D 中的代码。

当程序控制被传送到一致代码段，即使目标代码段的 DPL 小于 CPL，CPL 也不会改变。访问一致代码段是唯一一种可能导致当前代码段的 DPL 不等于 CPL 的情况（非一致代码段要求 CPL 和 DPL 相等）。因为没有发生 CPL 的切换，所以没有堆栈的切换。

一致代码段被用于支持应用程序但是却不需要访问受保护的资源的代码模块，比如数学函数库、异常处理例程。这些代码可能是内核的一部分，但是却允许低特权级的程序使用。在控制传送到更高特权级的一致代码段时保持 CPL 不变，避免了程序 CPL 提升到目标代码段 DPL 后去访问其他同特权级的非一致代码段的情况，阻止了低特权级的程序访问更多数据。

大部分代码段都是非一致的，毕竟只有一小部分情况下希望低特权级程序执行高特权级代码。对于一致代码段，只有同特权级的程序可以直接，不同特权级的程序只能通过调用门间接访问。

5.6 通过调用门访问代码段

调用门[3]用来实现不同特权级之间的控制传送，通常被操作系统或者其他使用特权级保护机制的特权程序中。

调用门描述符在 GDT 和 LDT 中，其他三种门描述符在 IDT 中。

5.6.1 IA-32 架构下的调用门

IA-32[4]架构下的调用门描述符结构如下：

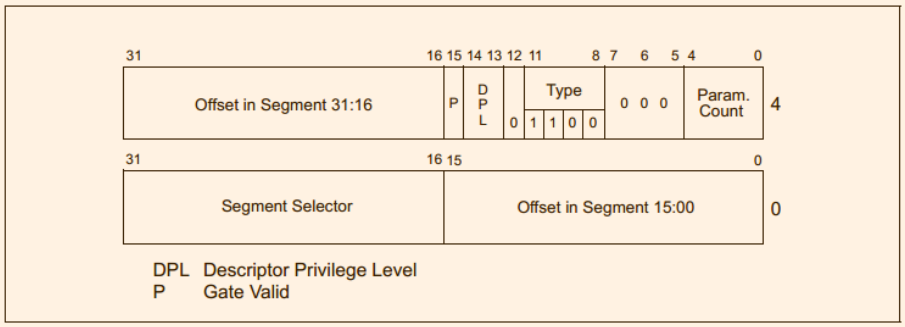

Figure 5-8. Call-Gate Descriptor

调用门描述符和其他门描述符结构基本相同。记录了目标代码段的选择子，目标过程在目标代码段中的偏移，调用门是否有效（是否在内存中），还记录了发生堆栈切换时需要从当前堆栈复制到目标堆栈上的参数个数。

参数个数对于 16 位调用门是 word 的个数，对于 32 位调用门是 doubleword 的个数。P flag 标识的是调用门是否有效，而不是调用门指向的代码段是否有效。如果访问 P flag 为 0 的调用门，会产生 #NP 异常。

[3] 调用门也用来实现 16 位代码段和 32 位代码段之间的控制传送，这一部分内容在 intel 手册第 21.4 节中，我没有阅读这部分内容，所以不表。

[4] 32 位时代时的架构叫 IA-32, 是 32 位处理器。到了 64 位时代，X86 处理器被拓展为 64 位，仍保持对之前 32 位处理器的兼容。64 位处理器加电后相当于 8086 处理器，开启保护模式后相当于 32 位处理器，这就是为什么 64 位处理器可以运行 32 位操作系统的原因。在开启相应的标志位，处理器就可以运行在 IA-32e 模式（真正的 64 位处理器）时，其中有 32 位兼容模式和 64 位模式两个子模式。就是因为有这两个子模式的存在，64 位操作系统才可以同时运行 32 位和 64 位模式。

5.6.2 64 位处理器 (IA-32e mode) 下的调用门

64 位架构（IA-32e 模式）下的调用门描述符结构如下：

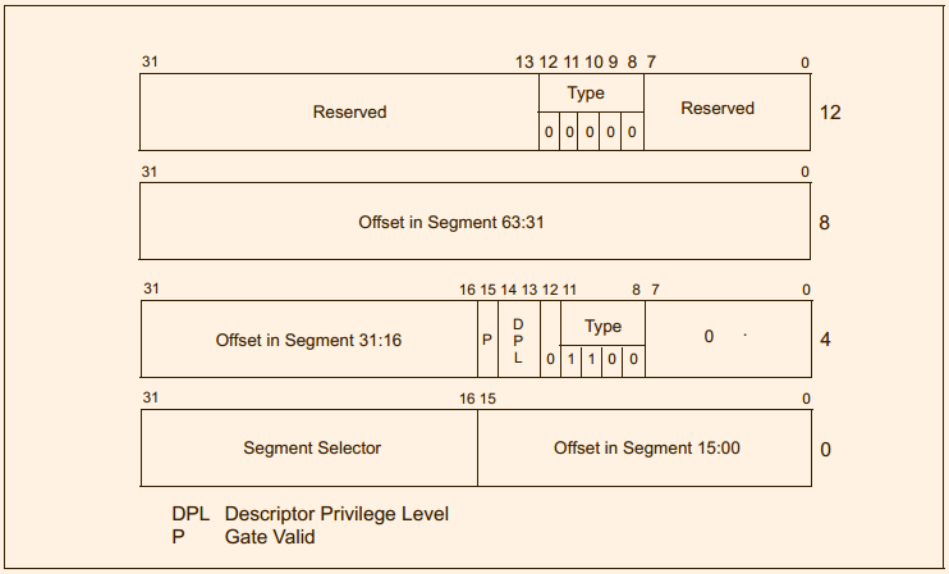

Figure 5-9. Call-Gate Descriptor in IA-32e Mode

为了容纳 64 位模式下的偏移，把原来 IA-32 架构中的调用门描述符大小翻倍; 为了兼容 32 位程序（32 位模式），保持了原先的结构。

在 IA-32e 模式下，已经没有了之前的 32 位的描述符，被重新定义成了 64 位。64 位调用门引用的代码段也必须是 64 位的（CS.L=1,CS.D=0），否则产生 #GP 异常; 把 64 位调用门描述符当成两个 32 位描述符会产生 #GP 异常。

在 64 位模式下访问调用门和 32 位模式下访问调用门没有多少区别，只是大小发生了改变，比如堆栈压入的寄存器变成 64 位。另外，在 64 位模式下不发生参数的复制。

5.6.3 通过调用门访问代码段的过程

通过调用门访问代码段的 JMP 和 CALL 格式和往常一样，但是给这两个指令提供的远指针（far pointer) 中只有选择子被使用，偏移被忽略（也不进行检查）。选择子用来在 GDT 或 IDT 中选择调用门。

调用门的权限检查的过程复杂不少，涉及：

- CPL
- RPL
- 调用门的 DPL
- 目标代码段的 DPL

- 目标代码段的 C flag

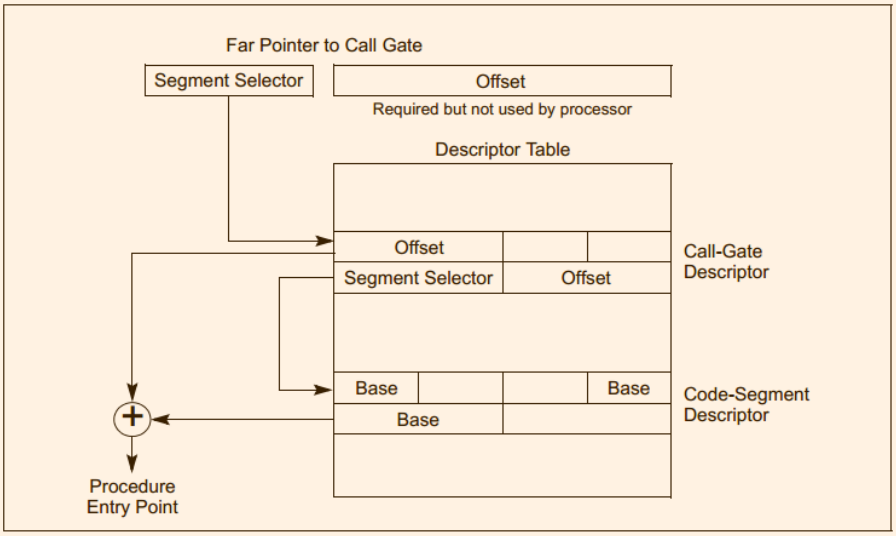

Figure 5-10. Call-Gate Mechanism

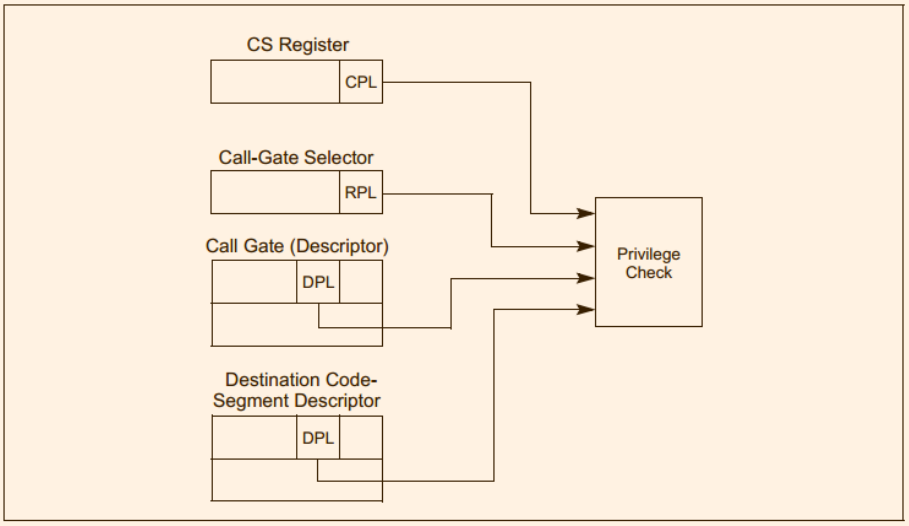

Figure 5-11. Privilege Check for Control Transfer with Call Gate

调用门实现的是低特权级到高特权级的访问，因此程序特权级应该小于等于目标代码段特权级，否则调用门就没有意义了;另一方面，为了访问调用门，CPL 和 RPL 应该小于等于调用门的 DPL。

使用 JMP 和 CALL 通过调用门访问代码段的具体规则如下：

Table 5-1. 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

CALL 指令可以实现从低特权级代码段到高特权级的非一致代码段的控制转移，而 JMP 指令不允许这样的操作。访问非一致代码段时 CPL 改变（变为目标代码段 DPL），访问一致代码段时 CPL 不改变。因此，从调用门访问非一致代码段时可能发生堆栈切换，访问一致代码段时不发生堆栈切换。

下面的例子说明通过调用门访问代码段时的特权级检查。

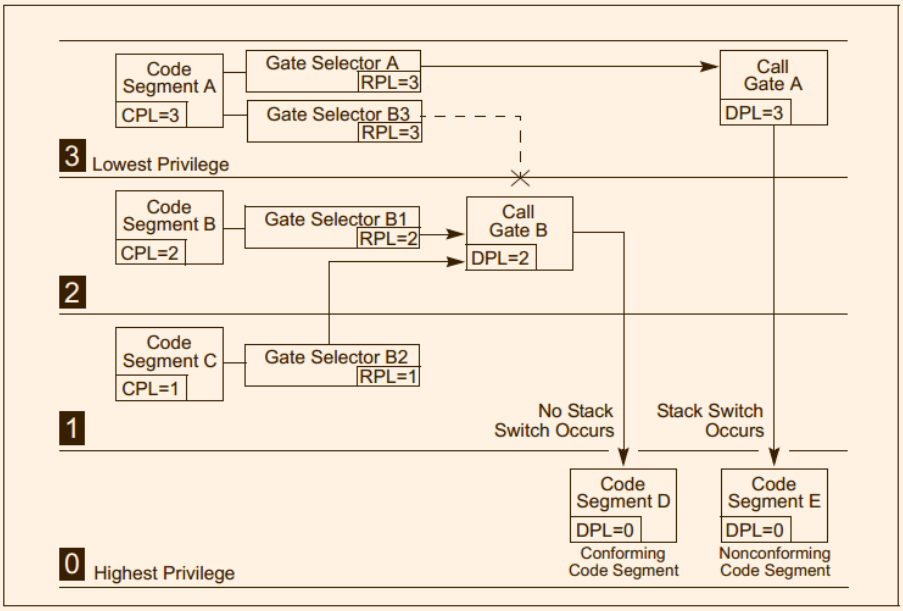

Figure 5-12. Example of Accessing Call Gates At Various Privilege Levels

- 代码段 A 的 CPL 大于调用门 B 的 DPL，所以无法访问调用门 B，自然也无法访问调用门指向的代码段。
- 代码段 A 的 CPL、选择子 A 的 RPL 都等于调用门 A 的 DPL，所以代码段 A 中的过程可以访问调用门 A; 调用门指向的代码段 E 的 DPL 小于代码段 A 的 CPL，所以可以使用 CALL 访问; 但是因为 CPL 不等于 E 的 DPL，不能够使用 JMP 访问。

5.6.4 堆栈切换

当程序通过调用门将控制传递到更高特权级的代码段时，处理器会自动将堆栈切换为目标特权级的堆栈。通过堆栈切换可以一定程度避免堆栈空间不足导致的程序崩溃和低特权级过程通过修改共享

堆栈的内容来修改高特权级过程。

每个任务都应该有 4 个堆栈，分别对应特权级 0 ~ 4，通常使用两个特权级的系统只需要为每个任务准备 2 个堆栈，分别对应特权级 0 和特权级 3。虽然一个任务拥有多个对应相应特权级的堆栈，但是任意时刻仅使用一个堆栈（只有一个 SS 和 SP）。正在使用的堆栈就是 SS：SP 指向的堆栈。所有的堆栈的指针都存储在任务对应的 TSS 中，并且在任务执行过程中不会修改。TSS 中记录的堆栈指针仅在堆栈切换时用来创建堆栈，当从被调用过程返回时新创建的堆栈会被废弃，这就类似 C 语言中的局部变量，也许从函数返回后它还存在，但是我们不应该再使用它。

为各个特权级创建对应的堆栈空间（可能还有堆栈段）并且把堆栈指针保存在 TSS 中是操作系统的责任。堆栈必须要足够大，尤其是在操作系统支持嵌套中断 (nested intetrupt) 时，否则后果是致命的。

5.6.4.1 IA-32 下的堆栈切换

当发生特权级切换时会进行堆栈切换，过程如下：

1. 使用目标代码段的 DPL（新 CPL）从 TSS 中选择新堆栈（指针）。
2. 读取指针，如果有任何发现任何段/表限长错误，产生 #TS 异常。
3. 检查堆栈所在段的特权级和类型，出错就产生 #TS 异常。
4. 暂存 SS 和 ESP 指针的当前值（当前堆栈）。
5. 加载新堆栈指针到 SS、ESP 中。
6. 将原来的 SS、ESP 压入新栈。
7. 从原来的堆栈拷贝 N 个参数到新栈中。N 是调用门中指定的参数个数。
8. 将返回地址（CS、EIP）压入新栈。
9. 从调用门中将目标地址的选择子和偏移加载到 CS、EIP 中。

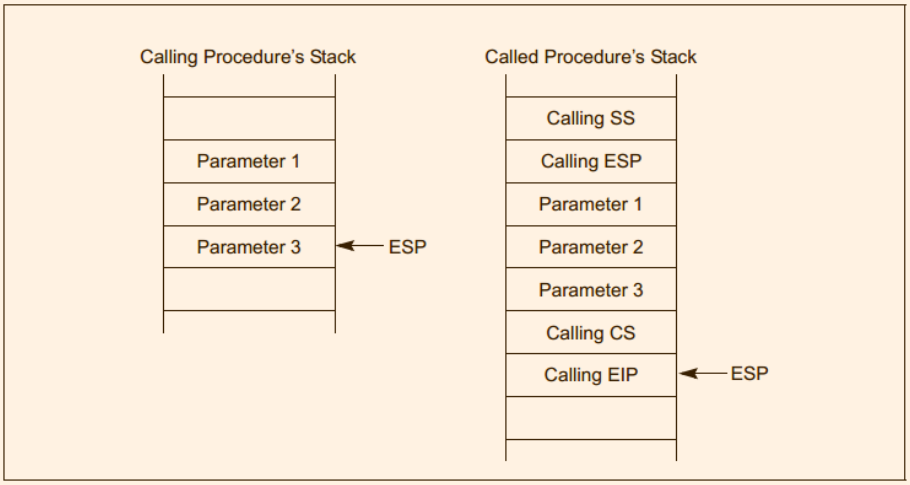

Figure 5-13. Stack Switching During an Interprivilege-Level Call

参数个数在调用门中占的比特决定了当发生堆栈切换时最多只能拷贝 31 个参数，如果需要更多的参数可以通过拷贝一个指向更多参数的指针或者通过访问旧堆栈实现。

5.6.4.2 IA-32e 模式下的堆栈切换 对于 32 位模式，堆栈切换的过程没有改变;64 位模式下的堆栈切换有所不同。

在 64 位模式下发生堆栈切换时，新的 SS（指向的）段描述符不被家在;64 位模式只从 TSS 中加载新的 RSP。新的 SS 被强制处理为 NULL（为了处理嵌套远转移），SS 选择子的 RPL 被强制设置为新的 CPL。旧的 SS 和 RSP 被保存在新堆栈中。新的

64 位模式下堆栈切换的布局[5]：

Table 5-2. 64-Bit-Mode Stack Layout After Far CALL with CPL Change

32-bit Mode			IA-32e mode		
Old SS Selector	+12		+24	Old SS Selector
Old ESP	+8		+16	Old RSP
CS Selector	+4		+8	Old CS Selector
EIP	0	ESP	RSP	0	RIP
< 4 Bytes >				< 8 Bytes >

在 64 位模式下压栈时是大小是 8 字节的，而不是 32 位时模式的 4 字节。64 位模式不执行 32 位模式中的参数拷贝，调用门中保存的参数个数也被忽略。

64 位模式中，在特定情形下 far RET 可以合法的加载 NULL SS。如果目标模式是 64 位模式并且目标 CPL 不等于 3,IRET 可以允许 SS 中是空选择子。如果被调用过程自身被中断，空 SS 会被压栈。在随后的 far RET 中，栈上的空 SS 作为标志告知处理器不要加载新的 SS 描述符。

5.6.4.3 从被调用过程返回 从被调用过程返回通过 RET 指令完成，RET 指令总是和 CALL 指令搭配使用，JMP 指令没有对应的返回指令。

对于近转移，因为在一个段中，所以返回时之进行段/表限长检查，而不进行特权级检查。

对于同一特权级的远返回 (far return)，处理器会进行特权检查。

对于不同特权级之间的远返回，只允许从高特权级向低特权级返回（返回的代码 DPL 大于 CPL），处理器会利用堆栈中保存的 CS 中的 RPL[6]来和 CPL 比较，如果 RPL 大于 CPL，就会发生跨特权级的返回。

从被调用过程返回到调用过程中时执行以下步骤：

1. 检查保存的 CS 寄存器中的 RPL 字段来决定返回时是否要发生特权级转换。
2. 将堆栈中保存的 CS、EIP 旧值重新加载到 CS、EIP 中（CPL 已经改变）。（会对代码段描述符和代码段选择子的 RPL 进行类型检查和特权级检查）

[5] intel 手册 5.8.5.1 节描述的有些含糊不清。说 64-bit mode 没有 32-bit mode 中的参数拷贝，但是表格中 32-bit mode 却没有拷贝参数。而且 32-bit mode 仅仅是 IA-32e mode 中的子模式，图中却并列。我认为 32 位模式和 IA-32 中的行为是相同的，图中的 IA-32e mode 应该特指 64 位模式，左边的 32-bit mode 仅仅用来和右边的 IA-32e mode 比较大小，而非真的 32 位模式下的布局。

[6] 没错，intel 手册就是这样说的，堆栈中保存的 CS 中的 RPL！CPL 是 CS 寄存器中的两个比特，而不是堆栈中保存的旧 CS 中的那两个比特。这里应该是指将 CS 中 CPL 位置的两个比特作为 RPL，进行类似于上文介绍的权限检查。

3. 如果 RET 指令有操作数 N，在已经弹出 CS、EIP 后向 ESP 递增 N 个字节，以弹出调用过程传递过来参数，这时 ESP 指向调用过程原来栈顶（如果不发生特权级/堆栈切换的话）。注意，如果是通过调用门调用的过程，要小心 RET 的操作数以字节为单位，而调用门中记录的参数个数以 word、doubleword 为单位，需要进行转换。

如果没有发生特权级切换，从调用过程返回的过程就完成了; 如果发生了特权级转换，还要进行以下步骤：

4. 将保存的 SS 和 ESP 重新加载到 SS、ESP 中，被调用过程的栈被废弃。加载堆栈指针时任何段/表限长错误都会产生 #GP 异常。堆栈的描述符也会进行类型和特权级的检查。
5. 如果 RET 指令包含操作数 N，在堆栈指针已经指向调用过程的堆栈后，递增 N 字节以清除调用过程传递给被调用过程的参数。ESP 不会进行段限长的检查，所以就算 ESP 超过了段限长，在下一次堆栈操作之前是无法发现的。
6. 检查 DS、ES、FS、GS 段寄存器。如果寄存器指向的段的 DPL 小于 CPL（除了一致代码），那么这个寄存器中的选择子会被设置为空选择子。

64 位模式的 RET 过程类似。
