中断和异常处理-读书笔记

2021113117-王宇轩

1. 中断和异常处理概述

中断（Interruptions）和**异常**（Exceptions）是这样一类事件：它们表示系统、处理器或当前正在执行的的程序中存在一种需要处理器注意的情况。中断和异常通常导致执行从当前程序或任务强制转移到一个特殊的软件程序或任务——**中断或异常处理程序**（an interrupt handler or an exception handler）。此过程即为中断和异常处理。

中断在程序执行中随机出现，作为对硬件信号的响应。系统硬件使用中断处理处理器外部的事件，如服务器外围设备的请求。软件也可以用INT ***n***指令生成中断。

异常出现于执行一条指令时处理器检测到各种错误状况时，如除零、保护违规、分页异常、内部机器故障。Pentium 4、Intel Xeon、P6系列和Pentium处理器的机器检查架构还允许处理器在检测到内部硬件错误和总线错误时生成机器检查异常。

当收到中断，或检测到异常，当前运行的程序或任务会被暂停，处理器执行中断或异常处理程序；完成后，处理器继续执行被中断的程序。被中断的过程或任务的恢复并不损失程序的连续性，除非不能从从异常中恢复或中断导致当前正在运行的程序被终止。

2. 有关中断和异常了解性的内容

2.1 中断和异常向量

为了帮助处理异常和中断，每个架构层面定义的异常和每个需要处理器特殊处理的中断条件都被分配一个唯一的标识号，称为**向量号**，作为处理器查询**中断描述符表（IDT）**的索引。IDT用于提供中断异常处理程序的进入点。

向量号的大小范围为0-255，其中0-31被Intel 64和IA-32系统架构保留，用于架构定义的中断或异常，这些保留向量号有的并不对应已定义的函数，但仍被保留，不允许用户使用。32-255之间的向量号被指定为用户定义的中断，不被架构保留，通常分配给外部I/O设备，以使这些设备能够通过外部硬件中断机制之一将中断发送给处理器。

Table 6-1. 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	RESERVED	Fault/ Trap	No	For Intel use only.
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	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU 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	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[5]
20	#VE	Virtualization Exception	Fault	No	EPT violations[6]
21-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

处理器的本地APIC通常连接到基于系统的I/O APIC。在这里，在 I/O APIC 的引脚可以通过系统总线（Pentium 4、Intel Core Duo、Intel Core 2、Intel® AtomTM 和 Intel Xeon 处理器）或 APIC 串行总线（ P6 系列和奔腾处理器）接收外部中断。 I/O APIC 确定中断向量号并将此编号发送到本地 APIC。当一个系统包含多个处理器时，处理器也可以通过系统总线（Pentium 4、Intel Core Duo、Intel Core 2、Intel Atom 和 Intel Xeon 处理器）或 APIC 串行总线（P6 系列和奔腾处理器）发送中断请求。

LINT[1:0] 引脚在不包含片上本地 APIC 的 Intel486 处理器和更早的奔腾处理器上不可用。这些处理器具有专用的 NMI 和 INTR 引脚。对于这些处理器，外部中断通常由基于系统的中断控制器 (8259A)生成，中断信号通过INTR引脚发出。

值得注意的是，处理器上的RESET#、FLUSH#、STPCLK#、SMI#、R/S# 和 INIT# 引脚可能会导致处理器中断发生，但这些中断不是由本章描述的中断和异常机制处理的。处理器是否

> 包含这些引脚取决于特定处理器上的实现。在处理器的数据手册中描述了处理器所包含的引脚功能。

有关**APIC**：

> APIC（Advanced Programmable Interrupt Controller）是一种高级可编程中断控制器，用于处理和分发计算机系统中的中断信号。它是现代计算机体系结构中常见的硬件组件之一。
>
> APIC的作用是管理系统中不同设备和组件之间的中断通信。它负责接收和响应来自硬件设备、外部中断引脚或其他处理器的中断请求，并将其分发给适当的处理器核心或处理线程。APIC还负责中断优先级的管理、中断屏蔽和中断向量的分配。

2.2 中断源和异常源

两类中断源：

- 可屏蔽硬件中断（Maskable Hardware Interrupts）
- 软件生成中断（Software-Generated Interrupts）

三类异常源：

- 处理器检测到的程序错误异常（Processor-detected program-error exceptions）
- 软件生成异常（Software-generated exceptions）
- 机器检查异常（Machine-check exceptions）

2.2.1 可屏蔽硬件中断

任何通过INTR引脚或本地APIC传送至处理器的外部中断都称为可屏蔽硬件中断，其中INTR方式包括架构内所有0-255中断向量号，APIC方式是16-255，APIC将经过其传送收到的中断0-15视为非法向量。EFLAGS寄存器中的IF标志允许将所有可屏蔽的硬件中断作为一个组进行屏蔽。

2.2.2 软件生成中断

INT ***n***指令允许软件内部通过提供一个中断向量作为操作数来生成中断，如`INT 35`指令强制对35号中断的处理程序进行隐式调用。

0-255向量都可以作为此指令的参数。然而，如果是处理器预定义的向量**NMI**（向量号2，不可屏蔽中断），即用正常方法生成NMI中断，处理器的回应会不同，NMI中断处理程序会被调用，但处理器的NMI处理硬件不会被激活。

软件用INT ***n***指令生成的中断不会被EFLAGS寄存器的IF位屏蔽。具体会在**2.5 开启和禁止中断**中加以说明。

2.2.3 处理器检测到的程序错误异常

处理器在执行应用程序、操作系统或executive时检测到程序错误时会生成一个或多个异常。Intel 64和IA-32架构为每个处理器可检测到的异常都定义了一个向量号。异常被分为故障、陷阱和中止，见2.3。

2.2.4 软件生成异常

INTO、INT 3和BOUND指令允许软件生成异常，但有限制。如果INT ***n***的参数是架构定义的异常之一，处理器会生成一个向量正确的中断（以访问异常处理程序）但并不会将错误码（error code）压进栈，即使相关的硬件生成的异常通常产生错误代码，也是如此。异常处理程序在处理时，依然会尝试从栈中弹出错误码，但因为没有，所以handler会弹出并丢弃EIP，导致返回到错误位置。

2.2.5 机器检查异常

P6系列和奔腾处理器提供内部和外部机器检查机制，用于检查内部芯片硬件和总线事务的操作。这些机制依赖于用户实现。当检测到机器检查错误时，处理器发出机器检查异常（向量号18）并返回错误代码。

2.3 异常的分类：故障、陷阱和中止

异常被分为这三类，取决于**它们被报告的方式，和引发异常的指令能否不失连续性地重新开始。**

- **故障**：指可以被修复，且一旦被修复，允许程序重新开始且不损失连续性的异常。当有故障报告时，处理器就把执行报错指令前的机器状态存储下来。故障处理程序的返回地址（CS和EIP寄存器中保存的内容）指向那条报错指令，而不是其下一条。
- **陷阱**：是在执行陷阱指令后立即报告的异常。陷阱允许程序或任务的执行不损失程序连续性地继续。陷阱中断程序返回地址指向陷阱指令的下一条指令。
- **中止**：中止异常不总是报告引起异常的指令的精确位置，且不允许引起该异常的程序或人物重新开始。中止用于报告严重错误，如硬件错误或系统表中不一致或非法值。

> 通常报告为故障的一个异常子集是不可重新启动的。此类异常会导致某些处理器状态丢失。例如，执行 POPAD 指令，其中堆栈帧越过堆栈段的末尾会导致报告错误。在这种情况下，异常处理程序发现指令指针 (CS:EIP) 已恢复，就好像 POPAD 指令未被执行一样。但是，内部处理器状态（通用寄存器）将被修改。这种情况被认为是编程错误。导致此类异常的应用程序应由操作系统终止。

2.4 程序或任务的重新执行

为允许在处理异常或中断后能够重新启动程序或任务，所有异常（除中止外）都保证能在**指令边界**上报告异常，所有中断都保证在指令边界上发生。

对于**故障**异常，返回指令指针（处理器生成异常时保存）指向故障指令，故当处理完返回时，故障指令被重新执行。这种重启普遍被用来处理访问操作数被屏蔽而生成的异常，此类故障最普遍的例子就是**缺页异常**（page-fault exception，#PF），出现于程序或任务引用的操作数所在的页不在内存的情况。缺页异常出现时，异常处理程序可以加载该页进内存，重启故障指令来恢复执行。为

了保证重启对当前程序或任务是透明的，处理器保存必要的寄存器和栈指针来允许故障指令执行前状态的重启。

对于**陷阱**异常，返回指令指针指向陷阱指令的下一条指令。如果在一条指令中检测到的陷阱**转移执行**，则返回指令指针会反应此转移。如执行`JMP`指令时检测到陷阱，返回指针会指向跳转目标指令，而不是`JMP`的下一条指令。所有陷阱异常允许程序和任务不损失连续性地重启。如，溢出异常（overflow exception）是一种陷阱异常，此时返回指针指向测试EFLAG.OF标志位的`INTO`指令的下一条指令，其陷阱处理程序解决溢出状况，返回时，程序或任务从INTO`指令的下一条指令继续执行。

中止异常不支持程序任务的可靠重启，中断处理程序用于收集中止异常出现时，关于处理器状态的诊断信息，之后尽可能优雅地关闭程序和系统。

中断严格支持被中断的程序任务不失连续性的重启。为中断保存的返回指令指针指向处理器收到中断的位置——中断边界的下一条将要执行的指令。如果刚刚执行的指令有一个重复前缀，那么处理器将在当前迭代结束时执行中断，并将寄存器设置为执行下一次迭代所需要的值。

P6系列处理器具有推测执行指令的能力，但这不会影响处理器接受并执行中断。中断发生在位于指令执行退出阶段的指令边界处，所以它们总是被插入处理器的"有序"指令流中。

除了P6系列之外，奔腾处理器和更早的 IA-32 处理器也能够执行不同数量的预取和初步解码。对于这些处理器来说，在指令实际"按顺序"执行之前，不会发出异常和中断信号。对于给定的代码示例，当代码在任何IA-32处理器系列上执行时，异常信号统一发生（除非定义了新异常或新操作码）。

2.5 开启和禁止中断

处理器可抑制某些中断的生成，取决于处理器的状态和EFLAGS寄存器中的RF和IF标志位。

2.5.1 屏蔽可屏蔽硬件中断

IF位可禁止可屏蔽硬件中断服务。若IF=0，处理器抑制传送至INTR引脚、或通过本地APIC的中断，使其无法生成外部中断请求；若IF=1，这些中断被正常处理。

IF位不影响传送到NMI引脚的不可屏蔽中断（NMI）[1]、或经由本地APIC以NMI模式传送的信息，也不影响处理器生成的异常。同EFLAGS中的其他标志，处理器回应硬件复位会清除IF。

可屏蔽硬件中断组包括向量号0-32的中断和异常，这一点可能引起疑惑。架构层面上，IF=1时，任何0-32号的中断都可以通过INTR引脚传送至处理器，任何16-32向量可通过本地APIC传送，随后处理器会生成一个中断并调用向量号所指向的中断或异常处理程序。因此，例如，通过INTR引脚调用缺页异常（用向量号14）是可能的；然而，其并非真正的缺页异常，而是一个中断。与INT ***n***指令相同，当通过INTR引脚生成一个向量号为异常的中断时，处理器不会将错误码入栈，因此异常处理程序可能不会正常执行。

处理器可以分别使用`STI`（set interrupt-enable flag）和`CLI`（clear...）指令设置或清除IF标志位。当且仅当 CPL 等于或小于 IOPL 时，才可以执行上述两条指令。如果在 CPL 大于 IOPL 时执行它们，则会生成一般保护异常 (#GP)。

除上述所述外，IF 标志还受以下操作的影响：

- PUSHF 指令将所有标志存储在堆栈中，而处理器可以在其中检查和修改这些标志。 之后，可以使用POPF 指令将修改后的标志加载回 EFLAGS 寄存器。
- 任务切换以及使用 POPF 和 IRET 指令时，处理器将加载 EFLAGS 寄存器。因此，上述情形也可用于修改 IF 标志的设置。
- 当通过中断门处理中断时，IF 标志自动清除，从而禁用可屏蔽硬件中断。 （但如果中断是通过陷阱门处理的，那么 IF 标志不会被清除。）

2.5.2 屏蔽指令断点

EFLAGS 寄存器中的 RF（resume）标志控制处理器对指令断点条件的响应。设置后，RF位将会阻止指令断点生成调试异常 (#DB)；清除RF位时，指令断点将产生调试异常。 RF 标志的主要功能是防止处理器在指令断点处进入调试异常循环。

2.5.3 任务切换时屏蔽异常和中断

为了切换至一个不同的任务段，软件通常使用一对指令，如：

```
MOV SS, AX
MOV ESP, StackTop
```

若在段选择子载入SS寄存器后，ESP寄存器被加载前出现异常或中断，在处理程序执行期间，这两个寄存器指向栈空间的逻辑地址是不一致的。为了防止此状况，处理器会在向SS的MOV和POP指令后，抑制中断、调试异常和单步陷阱异常，直到到达下一条指令后的指令边界[2]。其余的故障依然可生成。若用`LSS`指令修改SS寄存器（修改此寄存器的推荐方法）则不会发生此问题。

Q&A

1. **有关不可屏蔽中断NMI：**

> 不可屏蔽中断 (nonmaskable interrupt, NMI) 可以通过以下两种方式之一生成：
>
> - 外部硬件激活 NMI 引脚。
> - 处理器在系统总线（Pentium 4、Intel Core Duo、Intel Core 2、Intel Atom和Intel Xeon处理器）或APIC串行总线（P6系列和Pentium处理器）上使用交付模式NMI接收消息。
>
> 当处理器从以上两种来源中的任何一个接收到NMI时，处理器会立即通过调用中断向量号2指向的NMI处理程序来处理它。处理器还会调用某些硬件条件以确保不会接收到其他中断，屏蔽时间从接收到NMI中断开始，直到NMI处理程序完成执行结束。 当NMI中断处理程序正在执行时，处理器会阻止后续NMI的传送，直到下一次执行IRET指令为止。NMI的这种阻塞阻止了NMI处理程序的嵌套执行。我们建议用户通过中断门访问NMI中断处理程序以禁用可屏蔽硬件中断。处理器执行IRET指令将会解锁NMI，即使IRET指令会导致错误。例如，如果

> IRET指令在 EFLAGS.VM = 1 且 IOPL 小于 3 的情况下执行，那么会生成一般保护异常。在这种情况下，NMI将在异常处理程序被调用之前被取消屏蔽。

2. **有关指令边界（Instruction Boundary）：** 在计算机体系结构中，指令边界是指指令在内存中的对齐边界。在执行指令时，"指令边界"指的是处理器执行指令的边界或界限，即处理器在执行下一条指令之前的边界。

2.6 异常和中断的优先级

如果在指令边界上有多个异常或中断挂起，处理器会以预定顺序相应它们，如下表。

Table 6-2. Priority Among Simultaneous Exceptions and Interrupts

Priority	Description
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	Nonmaskable Interrupts (NMI) [1]
6	Maskable Hardware Interrupts [1]
7	Code Breakpoint Fault
8	Faults from Fetching Next Instruction - Code-Segment Limit Violation - Code Page Fault
9	Faults from Decoding the Next Instruction - Instruction length > 15 bytes - Invalid Opcode - Coprocessor Not Available
10 (Lowest)	Faults on Executing an Instruction - Overflow - Bound error - Invalid TSS - Segment Not Present - Stack fault - General Protection - Data Page Fault - Alignment Check - x87 FPU Floating-point exception - SIMD floating-point exception - Virtualization exception

3. 中断描述符表

中断描述符表（IDT）与每个异常/中断向量关联一个门描述符或任务，以解决对应的异常/中断。

3.1 IDT的构成

与GDT和LDTs相同，保护模式下的IDT是一个**8字节描述符**组成的数组。与GDT（首条目为空段选择子）不同的是，IDT的第一个条目可以是个描述符。为形成IDT的索引，处理器将异常或中断向量按 8 的倍数（门描述符中的字节数）进行缩放。因为向量只有256个，IDT不需包含256个以上的描述符，可以少于256，因为只有可能出现的中断和异常向量需要描述符。IDT中的所有空描述符槽都应该将描述符的persent标志设置为0。

IDT的基地址应对齐于8字节边界，以最大化缓存行填充的性能。界限值以字节表示，并与基地址相加以获得最末有效字节的地址，界限值为0，则有效字节数为1。因为IDT中的条目总是8字节长，界限值总比8的倍数小1，即8N+1。

3.2 获得中断处理程序的地址

首先，处理器获取IDT的地址。IDT可位于线性地址空间中的任何位置。处理器使用IDTR寄存器定位IDT，该寄存器保存 IDT的32位基地址和16位界限，如下图。

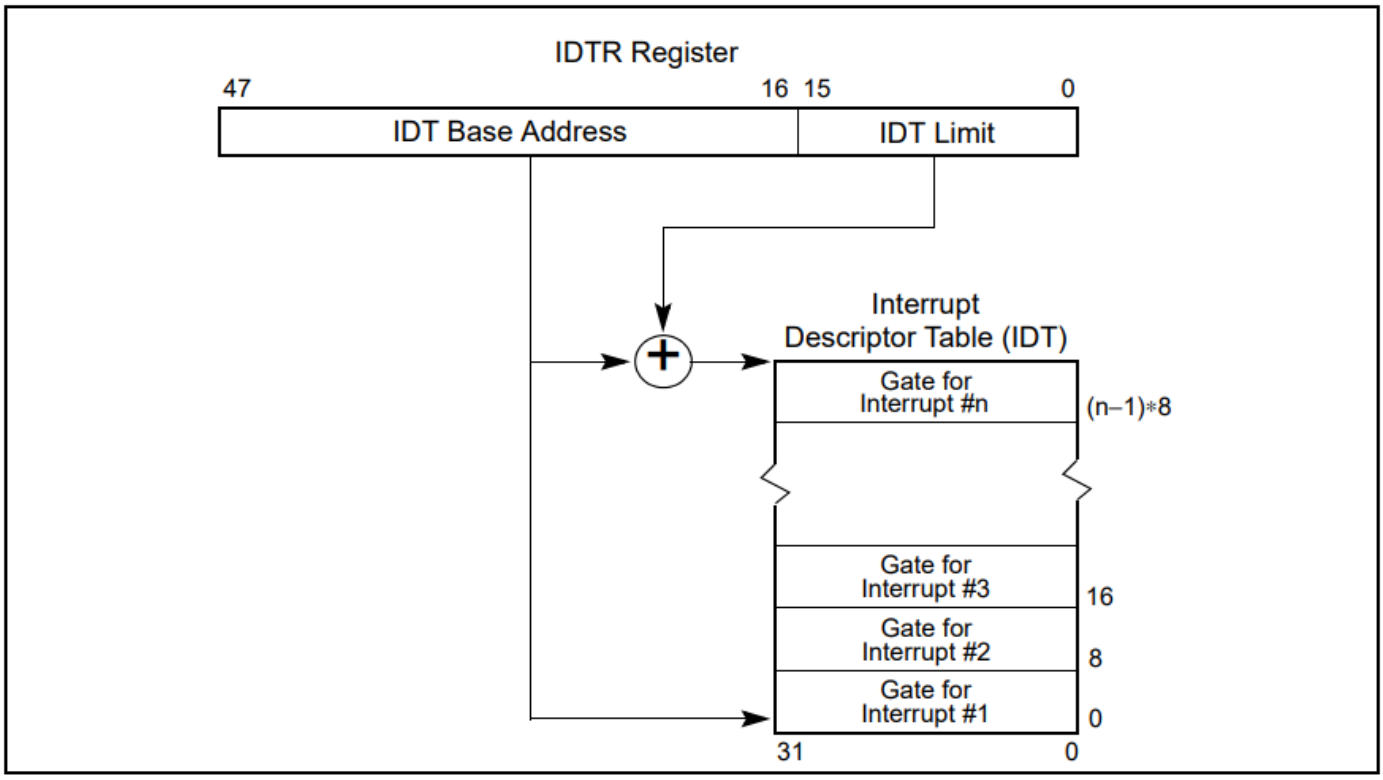

Figure 6-1. Relationship of the IDTR and IDT

确定IDT地址后，处理器用向量号作为索引，从IDT中获取中断描述符，进而确定调用中断处理程序的方式和其地址，将于**4. IDT 描述符**中详细介绍。如果向量尝试引用超出IDT限制的描述符，则会生成**一般保护异常 (#GP)**。

3.3 设置中断描述符表寄存器

可以使用`LIDT`（load IDT）和`SIDT`（store IDT）指令分别加载和存储 IDTR 寄存器的内容。`LIDT`指令将内存操作数中保存的IDT基地址和IDT表限制加载到 IDTR 寄存器中，只有在CPL为0时

才能执行。它通常是操作系统的初始化代码在创建IDT时使用。操作系统也可以使用它将一个IDT表更改为另一个。 `SIDT` 指令将存储在IDTR中的IDT基地址和表限制值复制到内存中。该指令可以在任何优先级的程序中执行。

4. IDT 描述符

IDT可包含以下三种**门描述符**：任务门描述符、中断门描述符、陷阱门描述符，其格式如下图。

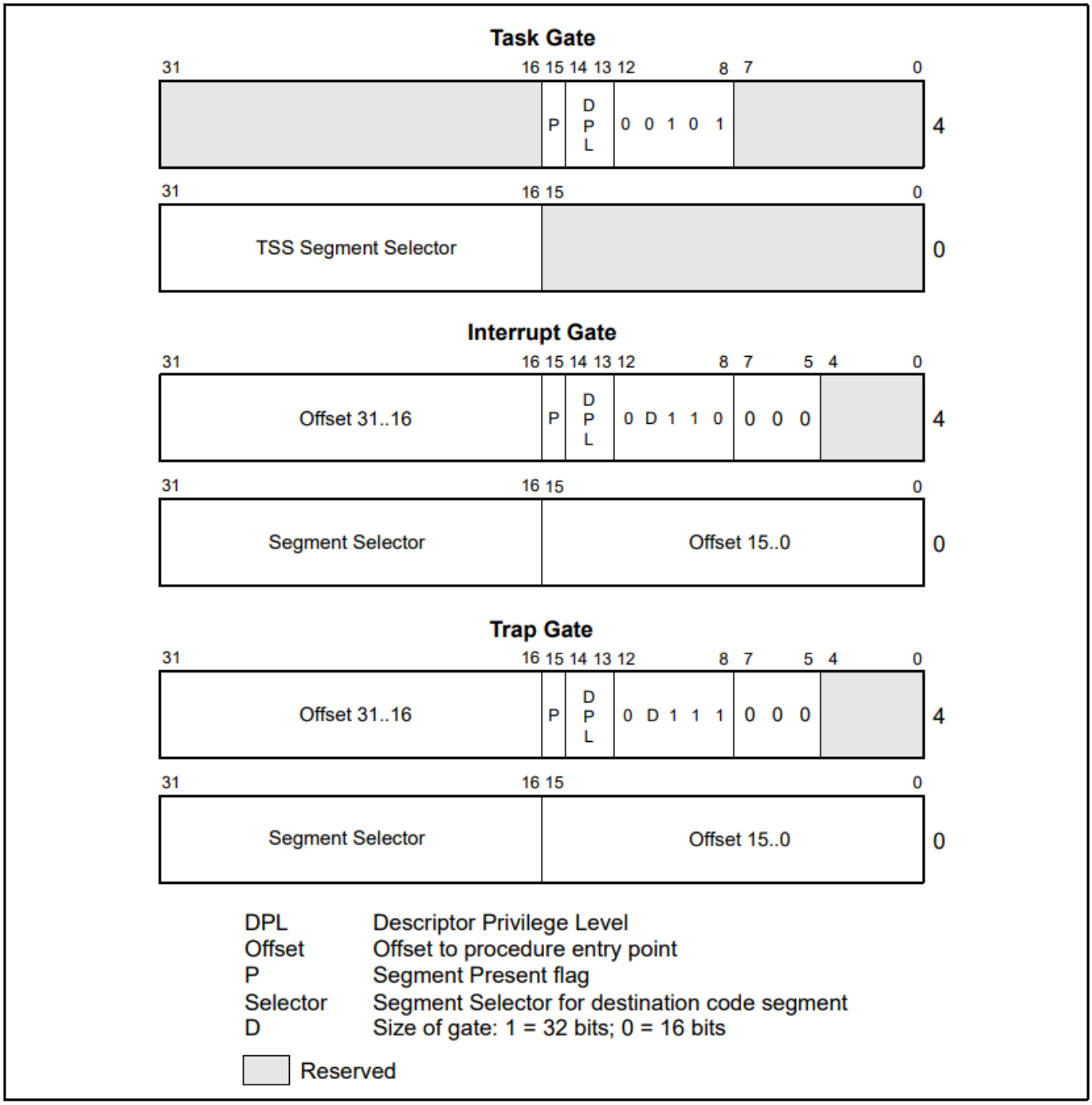

Figure 6-2. IDT Gate Descriptors

其中，IDT中的任务门和GDT、LDT中的是相同的，包含异常和/或中断处理**任务**的TSS的段选择子。

中断门和陷阱门与调用门很相似，包含一个远指针（段选择子和偏移量）用于处理器将程序执行转移至异常/中断处理程序代码段的处理程序过程。这些门在处理器处理 EFLAGS 寄存器中的 IF 标志

的方式上有所不同。

5. 中断与异常处理

5.1 中断过程调用的流程

处理器处理异常和中断处理程序的调用的方式与其处理用`CALL`指令对过程和任务的调用类似。当回应异常或中断时，处理器用向量号作为IDT中描述符的索引，若其指向中断门或陷阱门，处理器调用处理程序，方式与处理对调用门的`CALL`指令相似；若指向任务门，处理器执行任务切换至处理程序任务，方法与调用任务门的`CALL`指令类似。

中断门或陷阱门引用在当前执行任务的上下文中运行的异常或中断处理程序（见下图）。门的**段选择器**指向GDT或当前LDT中可执行代码段的**段描述符**。门描述符的**偏移字段**指向中断处理过程的**起始地址**。

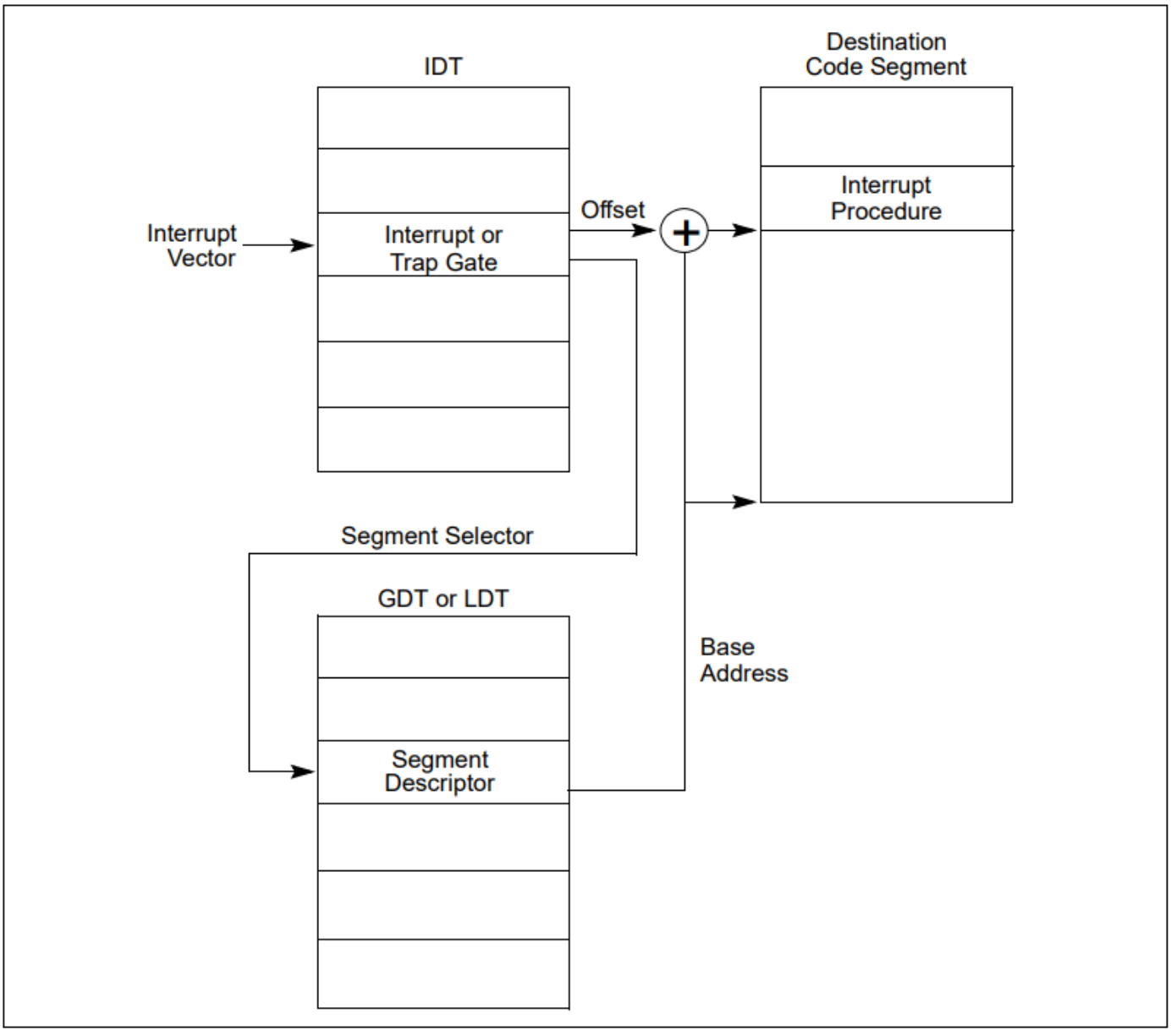

Figure 6-3. Interrupt Procedure Call

当处理器执行对异常或中断处理程序的调用时：

- 如果处理程序将在优先级较低的处理程序上执行，那么会发生**堆栈切换**。堆栈切换发生时： a. 从当前正在执行的任务的TSS中获取处理程序要使用的堆栈的段选择器和堆栈指针。在这个新堆栈上，处理器压入中断过程的堆栈段选择器和堆栈指针。 b. 然后，处理器将 EFLAGS、CS 和 EIP 寄存器的当前状态保存到新堆栈中（见下图）。 c. 如果异常导致错误码被保存，它会在 EIP 值之后被压入新堆栈。
- 如果处理程序将在优先级相同的处理程序上执行： a. 处理器将 EFLAGS、CS 和 EIP 寄存器的当前状态保存在当前堆栈中（见下图）。 b. 如果异常导致错误代码被保存，则将其压入 EIP 值之后的当前堆栈。

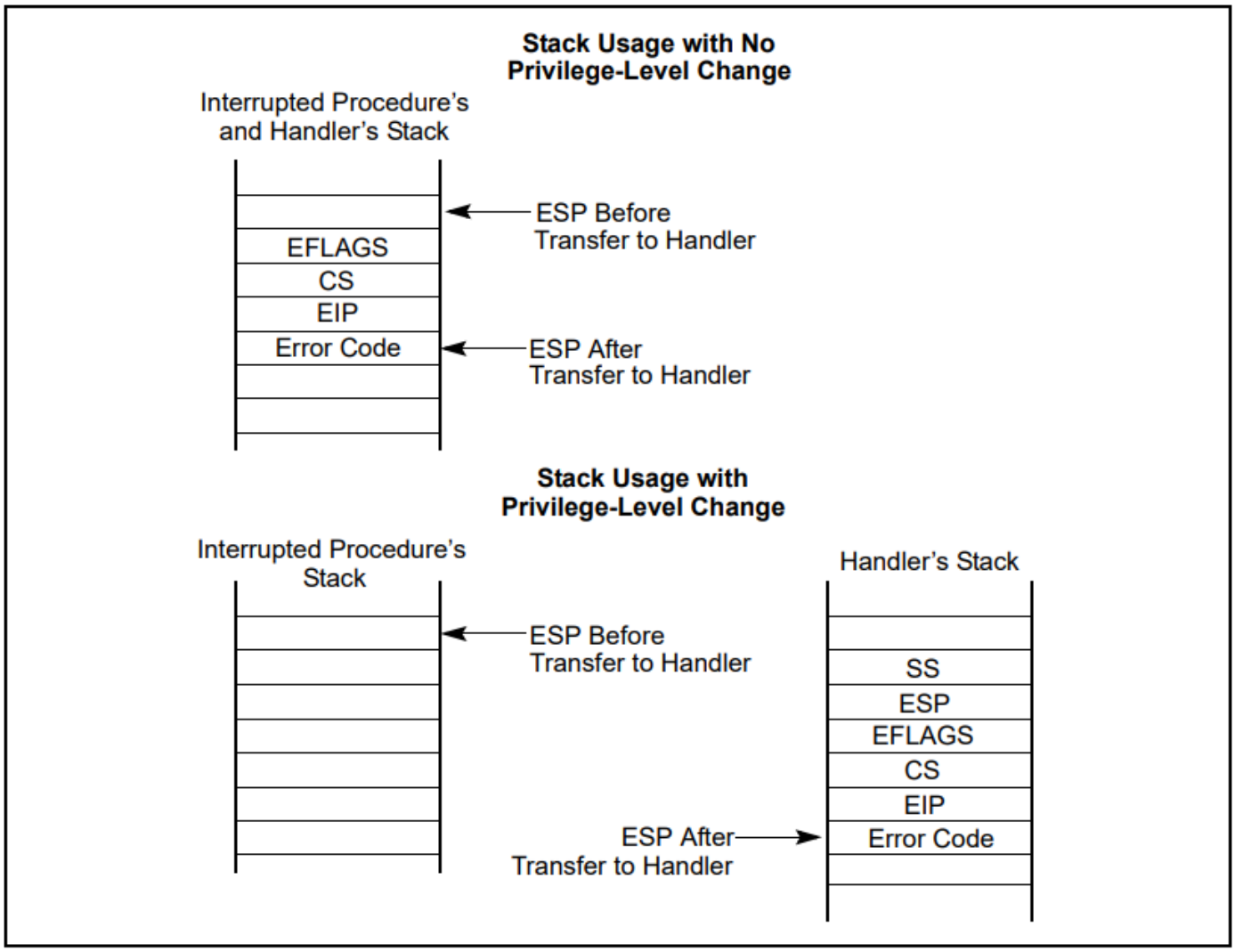

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

要从异常或中断处理程序**返回**，处理程序必须使用 IRET（或 IRETD）指令。 IRET 指令类似于 RET 指令，只是 IRET 指令将保存的标志恢复到 EFLAGS 寄存器中。只有当 CPL 为 0 时，EFLAGS 寄存器的 IOPL 字段才被恢复。只有当 CPL 小于或等于 IOPL 时，IF 标志才可以改变。

如果在调用处理程序过程时发生**堆栈切换**，则 IRET 指令会在返回时切换回被中断过程的堆栈。

5.2 异常和中断处理程序的保护

异常和中断处理程序的优先级保护类似于使用调用门调用时对于普通过程调用的保护。处理器不允许将执行转移到比**CPL**（Current Privilege Level，当前正在执行的代码所在的特权级别）优先级更低的代码段（特权级别数值更大）中的处理程序。

试图违反此规则会导致一般保护异常 (#GP)。异常和中断处理程序的保护机制在以下方面有所不同：

- 因为中断和异常向量没有**RPL**（Requested Privilege Level，代码段请求的特权级别），所以在对异常和中断处理程序的隐式调用时不检查RPL。
- 当且仅当使用INT *n*、INT 3 或 INTO 指令生成异常或中断时，处理器才会检查中断门或陷阱门的**DPL**（Descriptor Privilege Level，描述符特权级别）。此时 CPL 必须小于或等于门的 DPL。此限制可防止以优先级3（用户段）运行的应用程序或过程使用软件中断访问关键异常处理程序（比如缺页故障处理程序），因为这些处理程序位于更高优先级（数值小）的代码段。对于硬件生成的中断和处理器检测到的异常，处理器忽略中断和陷阱门的DPL。

因为异常和中断通常不会在可预测的时间发生，所以这些优先级规则有效地限制了异常和中断处理过程可以运行的优先级。处理器可以使用以下任一技术来避免优先级违规：

- 异常或中断处理程序可以放在一致的代码段中。此技术可用于只需要访问堆栈上可用数据的处理程序（例如，除法错误异常）。如果处理程序需要来自数据段的数据，则数据段需要可以从优先级级别 3 访问，这将使数据段不受优先级限制保护。
- 处理程序可以放置在优先级为 0 的不一致代码段中。该处理程序将始终可以运行，而忽视中断的程序或任务正在运行的CPL。

5.3 异常和中断处理过程的标志使用方式

当通过中断门或陷阱门访问异常或中断处理程序时，处理器会在将EFLAGS寄存器的内容保存到堆栈之后清除其中的TF标志。 （在调用异常和中断处理程序时，处理器也会在 EFLAGS 寄存器中的 VM、RF 和 NT 标志保存在堆栈中之后清除它们。）清除 TF 标志可防止指令跟踪影响中断响应。随后的IRET 指令将 TF（以及 VM、RF 和 NT）标志恢复为堆栈上 EFLAGS 寄存器中保存的值。

中断门和陷阱门之间的唯一区别是处理器处理 EFLAGS 寄存器中的 IF 标志的方式。当通过中断门访问异常或中断处理过程时，处理器清除 IF 标志以防止其他中断干扰当前中断处理程序。随后的 IRET 指令将 IF 标志恢复为其在堆栈上 EFLAGS 寄存器中保存的值。通过陷阱门访问处理程序时则不会影响 IF 标志。