中断、异常和系统调用

by zenhumany

2012-06-05——2012-07-01

目录

1. 概述

本章中所涉及的源代码全部来至 linux 内核 2.6.34。

中断通常被定义为一个事件，该事件改变处理器执行的指令顺序。这样的事件与 CPU 芯片内外部硬件电路产生的电信号相对应。

中断通常分为同步（synchronous）中断和异步（asynchronous）中断。

同步中断是当前指令执行时由 CPU 控制单元产生的，之所以称为同步，是因为该中断一般和当前运行的进程相关。

异步中断是由其他硬件设备依照 CPU 时钟信号随机产生的，该中断信号不一定和当前进程有关。

在 Intel 微处理器手册中，把同步和异步中断分别称为异常（exception）和中断。

中断是由间隔定时器和 I/O 设备产生的，例如用户的一次按键、网卡包的收发等。

异常是由程序的错误产生的（和当前运行的程序紧密相关），或者是由内核必须处理的异常条件产生的。

中断的分类：

- 可屏蔽中断：I/O 设备发出的所有中断请求（IRQ）都产生可屏蔽中断。
- 非屏蔽中断：只有几个危急事件才引起非屏蔽中断。

异常的分类：

当 CPU 执行指令时探测到的一个反常条件所产生的异常。可以进一步分为三组，这取决于 CPU 控制单元产生异常时保存在内核态堆栈 eip 寄存器中的值。

- 故障（fault）：比如缺页异常，通常可以纠正，一旦纠正，程序就可以在不失连贯性的情况下重新开始。保存在 eip 中的值是引起故障的指令地址。

- 陷阱（trap）：比如系统调用，在陷阱指令执行后立即报告，内核把控制权返回给程序后就可以继续它的执行而不失连贯性。保存在 eip 中的值是一个随后要执行的指令的地址。只有在没有必要重新执行已经终止的指令时，才触发陷阱。
- 异常终止（abort）：发生一个严重的错误，控制单元出了问题，不能在 eip 中保存引起异常的指令所在的确切位置。

多任务多进程操作系统的实现，都是基于中断提供的这种机制。
本章分析如下内容：
中断机制的初始化
中断的进入和返回
软中断

2. 中断机制的初始化

2.1 背景知识补充

2.1.1 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），用于指定内存分段管理所用系统表的基地址，如图 4-2 所示。处理器为这些寄存器的加载和保存提供了特定的指令。有关系统表的作用请参见 4.2 节"保护模式内存管理"中的详细说明。

内存管理寄存器

GDTR、LDTR、IDTR 和 TR 都是段基址寄存器，这些段中含有分段机制的重要信息表。GDTR、IDTR 和 LDTR 用于寻址存放描述符表的段。TR 用于寻址一个特殊的任务状态段（Task State Segment，TSS）。TSS 中包含着当前执行任务的重要信息。

（1）全局描述符表寄存器 GDTR

GDTR 寄存器中用于存放全局描述符表 GDT 的 32 位的线性基地址和 16 位的表限长值。基地址指定 GDT 表中字节 0 在线性地址空间中的地址，表长度指明 GDT 表的字节长度值。指令 LGDT 和 SGDT 分别用于加载和保存 GDTR 寄存器的内容。在机器刚加电或处理器复位后，基地址被默认地设置为 0，而长度值被设置成 0xFFFF。在保护模式初始化过程中必须给 GDTR 加载一个新值。

（2）中断描述符表寄存器 IDTR

与 GDTR 的作用类似，IDTR 寄存器用于存放中断描述符表 IDT 的 32 位线性基地址和 16 位表长度值。指令 LIDT 和 SIDT 分别用于加载和保存 IDTR 寄存器的内容。在机器刚加电或处理器复位后，基地址被默认地设置为 0，而长度值被设置成 0xFFFF。

（3）局部描述符表寄存器 LDTR

LDTR 寄存器中用于存放局部描述符表 LDT 的 32 位线性基地址、16 位段限长和描述符属性值。指令 LLDT 和 SLDT 分别用于加载和保存 LDTR 寄存器的段描述符部分。包含 LDT 表的段必须在 GDT 表中有一个段描述符项。当使用 LLDT 指令把含有 LDT 表段的选择符加载进 LDTR 时，LDT 段描述符的段基地址、段限长度以及描述符属性会被自动地加载到 LDTR 中。当进行任务切换时，处理器会把新任务 LDT 的段选择符和段描述符自动地加载进 LDTR 中。在机器加电或处理器复位后，段选择符和基地址被默认地设置为 0，而段长度被设置成 0xFFFF。

（4）任务寄存器 TR

TR 寄存器用于存放当前任务 TSS 段的 16 位段选择符、32 位基地址、16 位段长度和描述符属性值。它引用 GDT 表中的一个 TSS 类型的描述符。指令 LTR 和 STR 分别用于加载和保存 TR 寄存器的段选择符部分。当使用 LTR 指令把选择符加载进任务寄存器时，TSS 描述符中的段基地址、段限长度以及描述符属性会被自动加载到任务寄存器中。当执行任务切换时，处理器会把新任务的 TSS 的段选择符和段描述符自动加载进任务寄存器 TR 中。

2.1.2 X86 的TSS任务切换机制

segment descriptors 构建保护模式下的最基本、最根本的执行环境。system descriptors 则构建保护模式下的核心组件：

- TSS descriptor 提供硬件级的进程切换机制
- LDT descriptor 供进程使用多个 descriptor
- Gate descriptor 提供 processor 权限级别的切换机制

TSS 是一段内存区域，存放进程相关的执行环境信息。初始化的 TSS 是由用户提供，进程切换时的保存信息由 processor 执行。

2.1.2.1 TSS三个元素

2.1.2.1.1 TSS descriptor

这个 descriptor 属于 system descriptor 类型，它的 S (system）位是 0。

下面列出 TSS descriptors 的类型值：

0001： 16-bit TSS

0011： busy 16-bit TSS

1001： 32-bit TSS

1011： busy 32-bit TSS

以上是 x86 下的 TSS descriptor 类型，分为 16 和 32 位 TSS，x64 的 long mode 下 32 位的 TSS descriptor 将变为 64 位 TSS descriptor。

情景提示：

关于 TSS 的 busy 与 available 状态：

1、 TSS descriptor 的类型指明是 busy 与 available 状态。

2、TSS descriptor 的 busy 与 available 状态由 processor 去控制。即：由 processor 置为 busy 或 avaibable。 除了初始的 TSS descriptor 外。

TSS 的 busy 状态主要用来支持任务的嵌套。TSS descriptor 为 busy 状态时是不可进入执行的。同时防止 TSS 进程切换机制出现递归现象。

2.1.2.1.2 TSS selector以及 TR（Task Register）寄存器

TR 寄存器的结构与 segment registers 是完全一致的，即：由软件可见的 selector 部分与 processor 可见的隐藏部分（信息部分）构成。

TR.selector 与 CS.selector 中的 selector 意义是完全一样的。其 descriptor 的加载也是一样的。即： TR.selector 在 GDT / LDT 中索引查找到 TSS descriptor 后，该 TSS descriptor 将被加载到 TR 寄存的隐藏部分。当然在加载到 TR 寄存器之前，要进行检查通过了才可加载到 TR 寄存器。若加载 non-TSS descriptor 进入 TR 则会产生 #GP 异常。

2.1.2.1.3 TSS 块（Task Status Segment）

像 code segment 或 data segments 一样，最终的 TSS segment 由 TSS descriptor 来决定。 TSS descriptor 指出 TSS segment 的 base、limit 及 DPL 等信息。

TSS segment 存放 eflags 寄存器、GPRs 寄存器及相关的权限级别的 stack pointer （ss & sp）、CR3 等等信息。

2.1.2.2 TSS 机制的建立

对于多任务 OS 来说，TSS segment 是必不可少的，系统至少需要一个 TSS segment，但是现在的 OS 系统不使用 TSS 机制来进行任务的切换。

情景提示：

TSS 存在的唯一理由是：需要提供 0 ~ 2 权限级别的 stack pointer，当发生 stack 切换时，必须使用 TSS 提供的相应的 stack pointer。

但是：若提供空的 TSS segment，或者可以考虑以直接传递 stack pointer 的方式实现 stack 切换，即便是这样设计 processor 要读取 TSS segment 这一工作是必不可少的。

下面的指令用来建立初始的 TSS segment：

```
LTR word ptr [TSS_Selector] /* 在 [TSS_selector] 提供 TSS selector */
或：LTR ax    /* 在 ax 寄存器里提供 TSS selector   */
```

ltr 指令使用提供的 selector 在 GDT / LDT 里索引查找到 TSS descriptor 后，加载到 TR 寄存器里。初始的 TSS descriptor 必须设为 available 状态，

否则不能加载到 TR。processor 加载 TSS descriptor 后，将 TSS descriptor 置为 busy 状态。

2.1.2.3 TSS 进程切换的过程

当前进程要切换另一个进程时，可以使用 2 种 selector 进行：使用 TSS selector 以及 Task gate selector（任务门符）。

如：

```
call 0x2b:0x00000000            /* 假设 0x2b 为 TSS selector */
call 0x3b:0x00000000            /* 假设 0x3b 为 Task-gate  selector */
```

TSS 提供的硬件级进程切换机制较为复杂，大多数 OS 不使用 TSS 机制，是因为执行的效能太差了。上面的两条指令的 TSS 进程切换的过程如下：

2.1.2.3.1 使用 TSS selector

```
call 0x2b:0x00000000            /* 0x2b 为 TSS selector */
```

这里使用 jmp 指令与 call 指令会有些差别，call 允许 TSS 进程切换的嵌套，jmp 不允许嵌套。

- processor 使用 TSS selector （0x2b） 在 GDT 索引查找第 5 个 descriptor
- processor 检查找到的 descriptor 类型是否是 TSS descriptor，不是的话将产生 #GP 异常。是否为 available TSS，若目标 TSS descriptor 是 busy 的话，同样将产生 #GP 异常。
- processor 进行另一项检查工作：权限的检查，CPL <= DPL 并且 selector.RPL <= DPL 即为通过，否则产生 #GP 异常。
- 当前进程的执行环境被保存在当前进程的 TSS segment 中。
  - 在这一步里，此时还没发生 TSS selector 切换，processor 把当前进程的环境信息保存在当前的 TSS segment 中。
- 这里发生了 TSS selector 切换。新的 TSS selector 被加载到 TR.selector，而新的 TSS descriptor 也被加载到 TR 寄存的隐藏部分。
  - 这里，processor 还要将旧的 TSS selector 保存在当前的 TSS segment（新加载的 TSS）中的 link 域。这个 TSS segment 中的 link 其实就是 old TSS selector 域，用来进程返回时获得原来的 TSS selector 。从而实现任务的嵌套机制。

- processor 将当前 eflags 寄存器的 NT（Nest Task）标志位置为 1，表明当前的进程是嵌套内层。

- processor 从当前的 TSS segment 取出新进程的执行环境，包括：各个 selector registers（segment registers）、GPRs、stack pointer （ss & sp）、CR3 寄存器以及 eflags 寄存器等。
  - 在这一步，在加载 selectors 进入 segment registers 之前，还必须经过相关的 selector & descriptor 的常规检查以及权限检查。通过之后才真正加载。否则同样产生 #GP 异常。
  - processor 还要做另一项工作，就是：将新进程的 TSS descriptor 置为 busy 状态。使得新进程不能重入。
- processor 从当前的 CS:RIP 继续往下执行，完成这个 TSS 进程的切换。这个 CS: RIP 就是新加载的新进程的 cs : rip

从上面的过程可以看出，使用 TSS 进程切换机制异常复杂，导致进程切换的效能太差了。比使用 call gate 以及 trap gate 慢上好多。若当中发生权限的改变，还要发生 stack 切换。进程的返回同样复杂。同样需要这么多步骤，总结来说，主要的时间消耗发生在新旧进程的信息保存方面。

2.1.2.3.2 使用 task gate selector

另一种情况是使用 task gate selector 进行 TSS 进程切换，使用 task gate selector 除了可以 call/jmp 外，还可用在中断机制上，下面的两种情况：

```
call 0x3b:0x00000000          /* 0x3b 为 task gate selector */
或：int 0x3e                  /* 假设 0x3e 是 task gate 的向量 */
```

这两种情形差不多，除了一些细微的差别外，不过是触发的机制不同而已。

processor 在 IDT 索引查找到的是一个 task gate descriptor，这个 task gate descriptor 中指出了目标的 TSS selector 。processor 从 task gate descriptor 里加载 TSS selector，剩下的工作和使用 TSS selector 进行切换进程是一致。

processor 访问 task gate descriptor 仅需对 task gate descriptor 作出权限检查：CPL <= DPL 并且 RPL <= DPL。在这里不需要作出对 TSS descriptor 的 DPL 权限进行检查。

gate descriptor 机制提供了一层间接的访问层，主要用来控制权限的切换。

实际上：

对于 call 0x2b:0x00000000 这条指令，processor 并不知道这个 selector 是 TSS selector 还是 task gate selector，在查找到 descriptor 后，processor

查看这个 descriptor 的 type 才能确定是 TSS selector 还是 task gate selector。

最后需注意：

- 使用 jmp 指令进行转移，processor 不会将 eflags 中 NT 标志位置
- 使用中断机制下的 TSS 进程切换，processor 将不作任务的权限检查动作。

2.1.2.4 TSS 进程的返回

从 TSS 机制切换的进程在执行完后使用 iret 指令返回原进程进，同样会发生新旧 TSS segment 的更新动作。

进程通过使用 ret 返回时，processor 将不会从嵌套内层返回到的嵌套外层进程，也就是不会返回原进程。processor 对 ret 指令的处理，只会从 stack pointer（ss : esp） 取返回地址。

对于使用进程使用了 iret 中断返回指令时：

- processor 将会检查当前的 eflags.NT 标志位是否为 1，也就是检查当前进程是否处于嵌套的内层。
- eflags.NT = 1，processor 将从当前 TSS segment 中的 link（old TSS selector）域中取出原来进程的 TSS selector。
- processor 将不作任何的权限检查，TSS selector 被加载到 TR.selector，TSS descriptor 同时被加载到 TR 的隐藏部分。
- processor 将清除当前的 eflags.NT 为 0。若是使用 ret 指令返回的，processor 是不会清 eflags.NT 为 0
- 从 TSS segment 中加载新的进程执行环境，从新的 CS:EIP 处继续执行。 将原来的 TSS descriptor 重新置为 available 状态，使得可以再次进入。

由上可得，processor 遇到 ret 指令时，是不会对 eflags.NT 进行检查的。而使用 iret 指令，processor 将对 eflags.NT 进行检查。

2.2 X86 CPU对中断的硬件支持

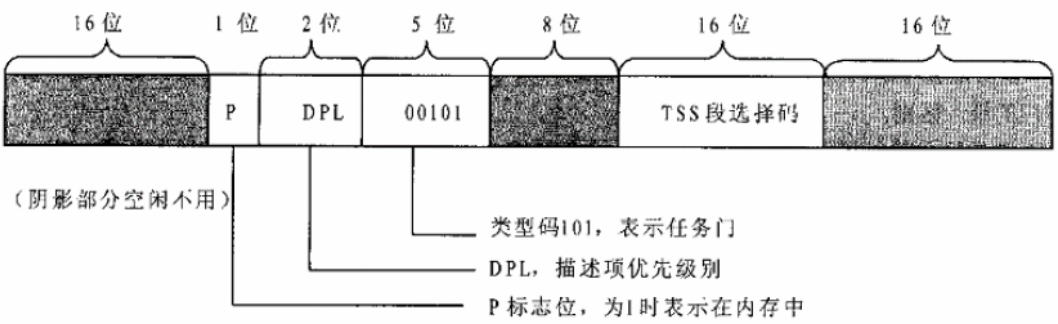
Intel X86 CPU 支持 256 个不同的中断向量，这一点一致未必。早期的 X86 CPU 的中断响应机制非常的原始、简单。在实地址模式中，CPU 把内存中从 0 开始的 1K 字节作为中断向量表。表中的每一项占据 4 个字节，由两个字节的的段地址和两个字节的位移组成。这样构成的地址便是响应的中断服务的程序的入口地址。这与 16 为的实地址模式中的寻址方式也是一致的。但是，在这样的机制上是不能构筑现代意义的操作系统的。即使把 16 为寻址改为 32 位，即使实现了页式存储，也是无济于事的。原因在于，这种机制被没有提供空间切换，或者说运行模式切换的手段。

因此，CPU 在实现保护模式时，对 CPU 的中断响应机制做了大幅度的修改。

首先，中断向量表中的表项重单纯的入口地址改成了比较复杂的描述符，称为“门”，意思是当中断发生时，必须穿过这些门才可以到达相应的服务程序。但是，这样的门并不是只为中断而设计的，只要想切换 CPU 的运行状态，即改变其运行级别，就要通过一道门。而从用户态进入系统态的途径也并不是只限于中断，还可以通过子程序调用指令 call 或者转移指令 jmp 来达到目的。而且，当中断发生时不但可以切换 CPU 的运行状态并转入中断服务程序，还可以安排进行一次任务切换（所谓“上下人”切换），立即切换到另一个进程。

按不同的用途和目的，CPU 中一共有四种门，即任务门（task gate）、中断门（interrupt gate）、陷阱门（trap gate）以及调用门（call gate）。其中任务门外其他三种门的结构基本相同，不过调用门并不是与中断向量表相联系的。

先看任务门，其大小为 64 位，如下图所示：

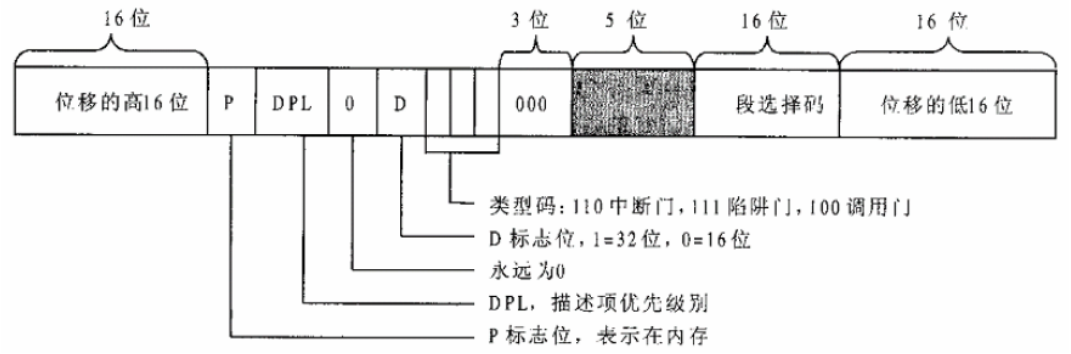


TSS 端选择子的作用和段寄存器 CS、DS 相似，通过 GDT 或 LDT 指向特殊的“系统段”中的一种，称为“任务状态段”TSS。

TSS 实际上是一个用来保存任务运行“现场”的数据结构，其中包括 CPU 中所有与具体进程有关的寄存器的内容（包含页面目录指针 CR3），还包括三个堆栈指针。中断发生时，CPU 在中断向量表中找到对应的表项，如果此表项是一个任务门，并且通过了优先级别的检查，CPU 就会将当前的任务的运行现场保存在相应的 TSS 中，并将任务门所指向的 TSS 作为当前任务，将其内容装

入 CPU 中的各个寄存器，完成一次任务的切换。为此目的，CPU 中有增设了一个“任务寄存器”TR，用来执行当前任务的 TSS。CPU 中并不采用任务门作为进程切换的手段。通过任务门切换到一个新的进程并不是唯一的途径，在程序中使用 JMP 指令或者调用门也可以达到同样的效果。

其余三种门的结构基本相同。



三种门之间的不同之处在于 3 为的类型码。与任务门相比，不同之处在于：任务门中不需要使用段内位移，而中断门、陷阱门和调用门则都要指向一个子程序，所以必须结合段选择码和段内位移。

2.3 运行级别的切换

2.3.1 优先级检测

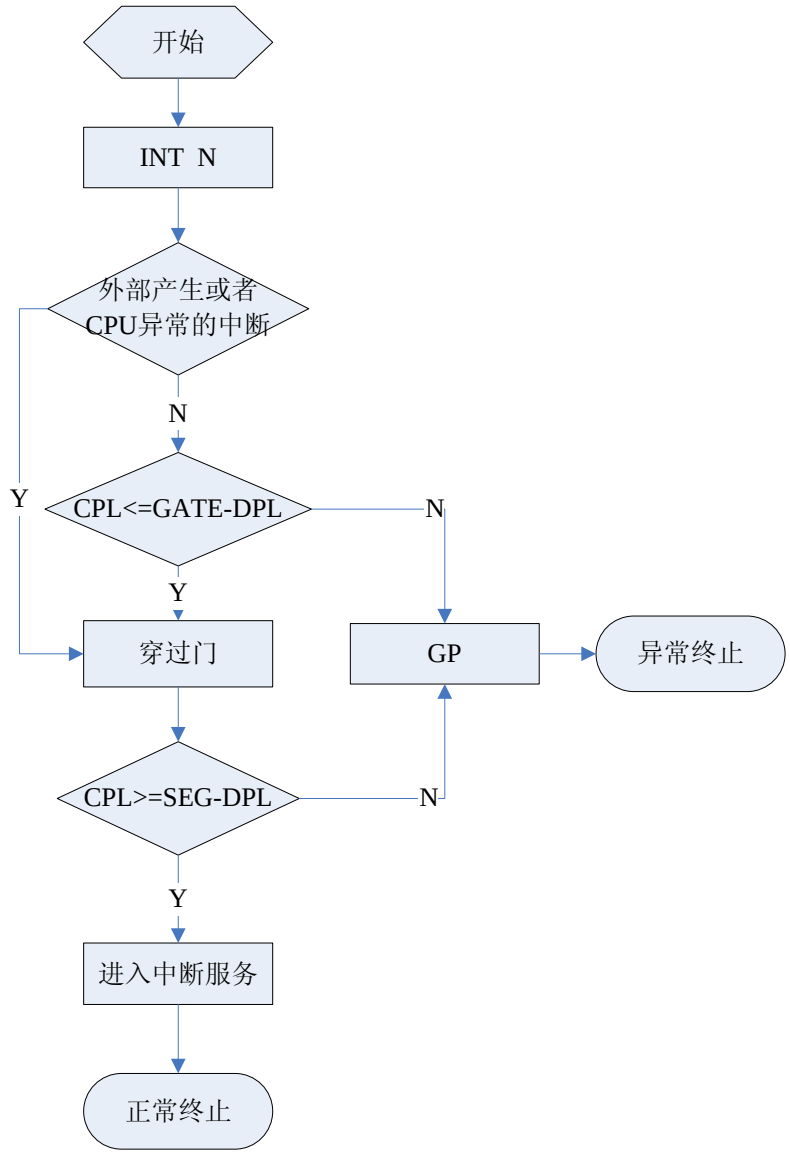
当通过一条 INT 指令进入中断服务时，在指令中给出一个中断向量。CPU 先根据该向量在中断向量表中找到一扇门。然后，就要将这个门的 DPL 与 CUP 的 CPL 比较，CPL 必须小于或者等于 DPL，也就是优先级别不能低于 DPL。如果中断是由外部产生的话，就要免去这一层检验。穿过中断门之后，还要进一步将目标代码段描叙符中的 DPL 与 CPU 比较，目标段的 DPL 必须小于或等于 CPL。也就是通过中断门时只允许保存或提升 CPU 的运行级别。

CPL：CPU 的运行基本

GATE-DPL：门描叙符中给定的(优先级别)，这个级别的设定可以控制在用户态是否可以穿越该门。

SEG-DPL：门描述符指向的段，段中给出的优先级别。

中断时优先级检测过程如下图所示：

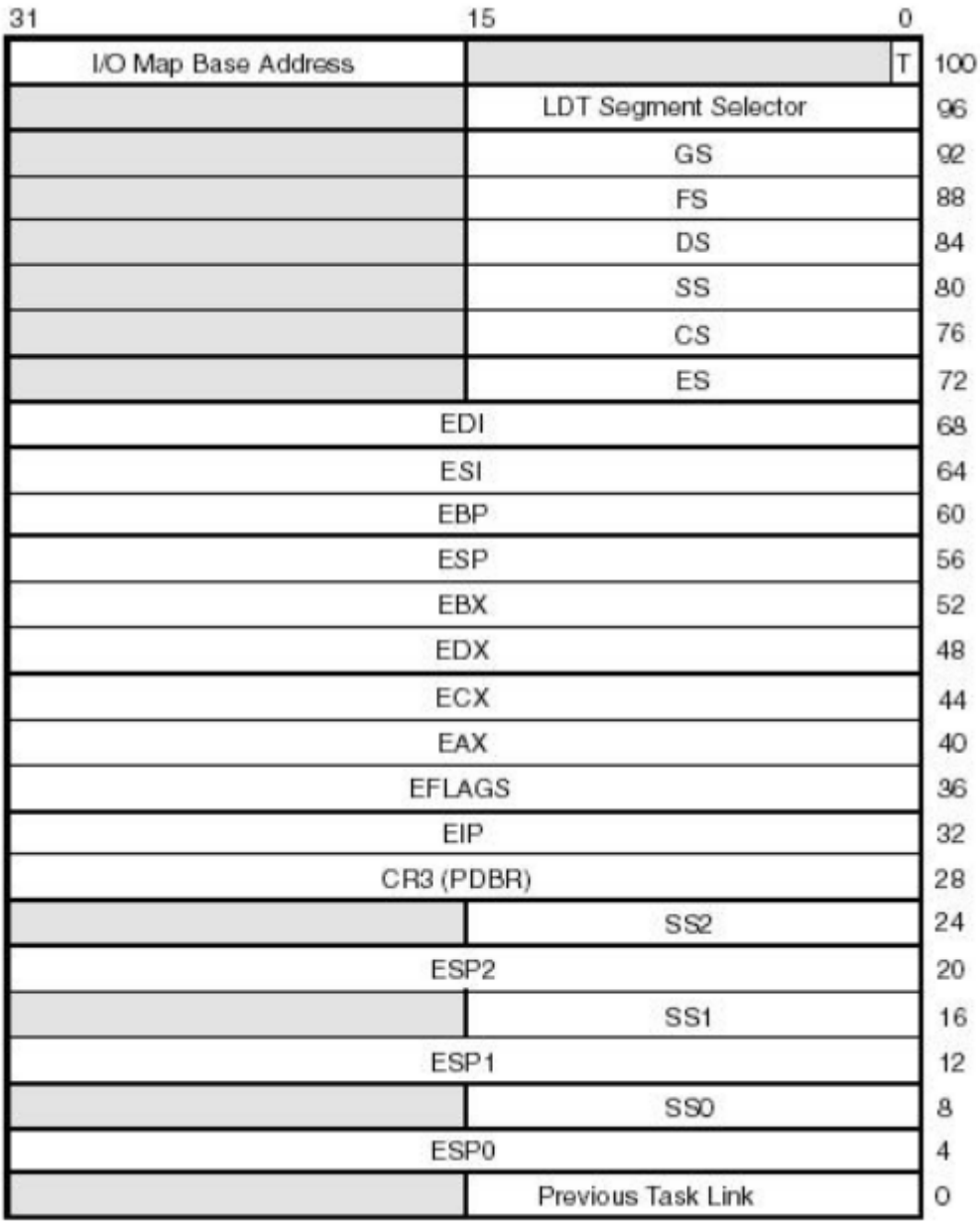


2.3.2 堆栈变化

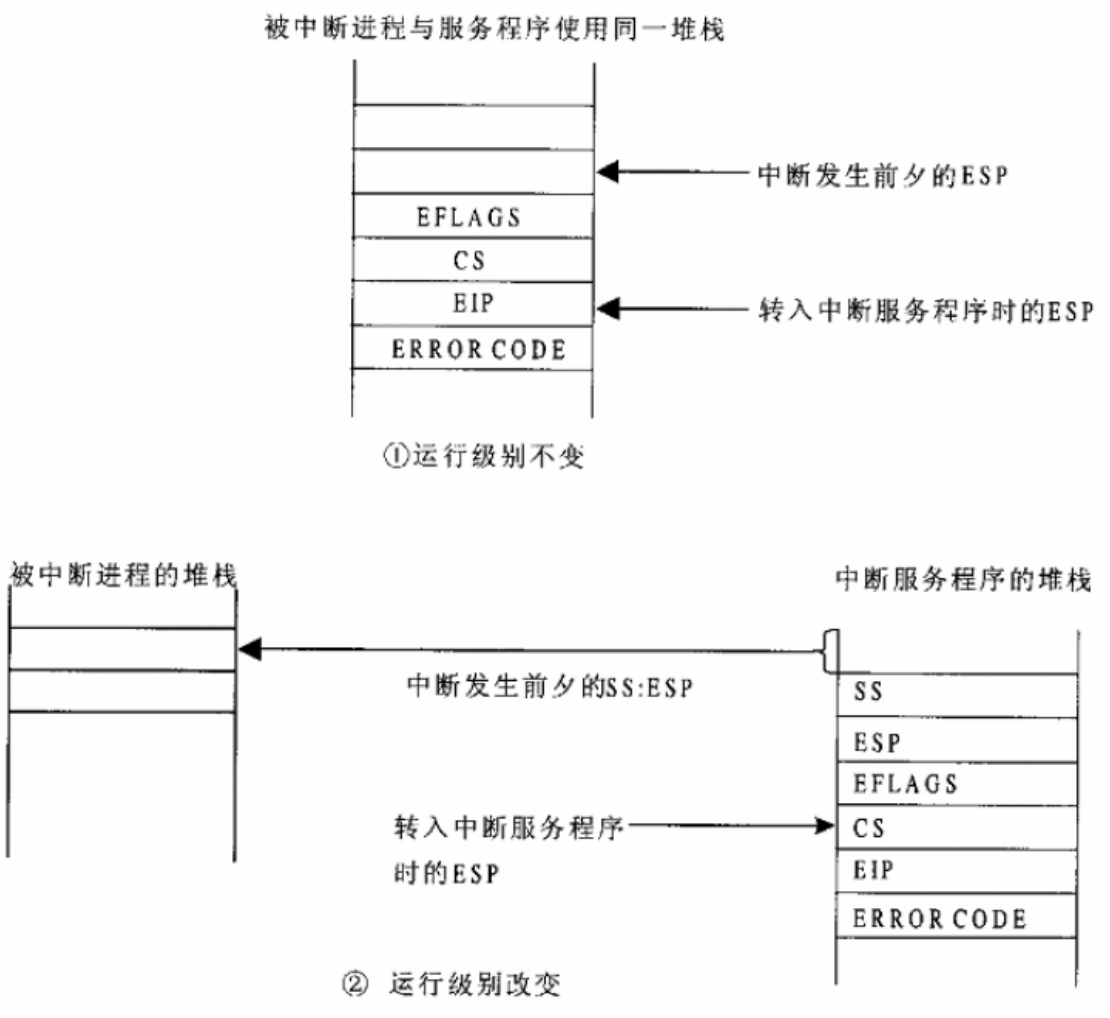
进入中断程序时，CPU 要将当前的 EFLAGS 寄存器的内容及返回地址压入堆栈，返回地址是由段寄存器 CS 和取指寄存器 EIP 的内容共同组成。如果中断是由异常引起的，还要将一个表示异常原因的出错码也压入堆栈。进一步，如果中断服务发生运行级别的切换，那就要引起堆栈的切换。

TSS 结构中除去所有的常规的寄存器（包括 SS 和 ESP）外，还有三对额外的堆栈指针（SS 加 ESP）。这三个额外的堆栈指针分别用于 CUP 运行于 0、1、2 级别时。所以，CPU 根据 TR 的内容找到当期的 TSS 结构，并根据目标代码的 DPL，从 TSS 中取出新的 SS 和 ESP，并装入 SS 和 ESP 寄存器，达到切换

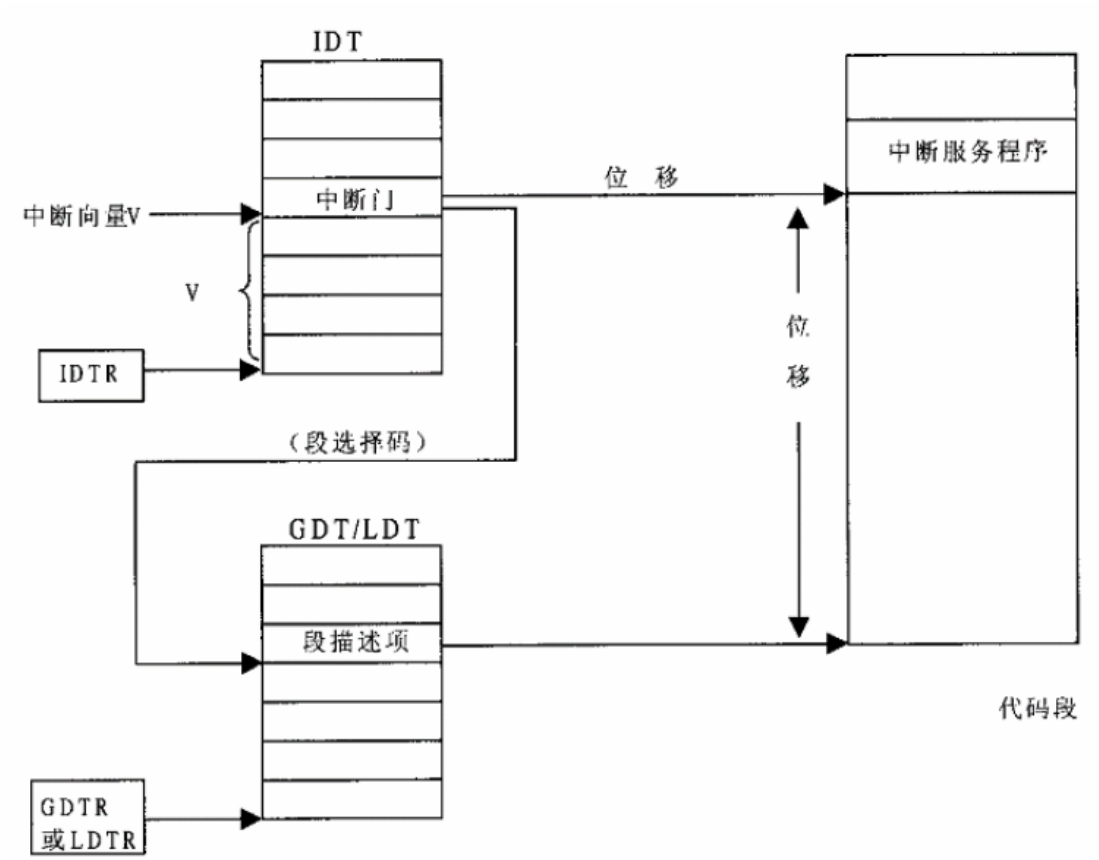
堆栈的目的。这种情况下，CPU 不断要将 EFALGS、返回地址以及出错码压入堆栈，还要将原先的堆栈指针也压入堆栈（新堆栈）。



TSS 结构


被中断进程与服务程序使用同一堆栈
中断发生前夕的ESP
EFLAGS
CS
EIP
转入中断服务程序时的ESP
ERROR CODE
①运行级别不变
被中断进程的堆栈
中断服务程序的堆栈
中断发生前夕的SS:ESP
SS
ESP
EFLAGS
转入中断服务程序
时的ESP
CS
EIP
ERROR CODE
② 运行级别改变

中断堆栈变化



中断机制示意图

2.4 中断和异常的硬件处理

现在开始分析 CPU 控制单元如何处理中断和异常。假定内核已被初始化，因此，CPU 在保护模式下运行。当执行了一条指令后，CS 和 eip

这对寄存器包含下一条将要执行的指令的逻辑地址。在处理那条指令之前，控制单元会检查在运行前一条指令时是否已经发生了一个中断或异常。如果发生了一个中断或异常，那么控制单元执行下列操作：

1. 确定与中断或异常关联的向量 i （0 ≤ i ≤ 255）。

2. 读由 idtr 寄存器指向的 IDT 表中的第 i 项（在下面的分析中，我们假定 IDT 表项中包含的是一个中断门或一个陷阱门）。

3. 从 gdtr 寄存器获得 GDT 的基地址，并在 GDT 中查找，以读取 IDT 表项中的选择符所标识的段描述符。这个描述符指定中断或异常处理程序所在段的基地址。

4. 确信中断是由授权的（中断）发生源发出的。首先将当前特权级 CPL（存放在 cs 寄存器的低两位）与段描述符（存放在 GDT 中）的描述符特权级 DPL 比较，如果 CPL 小于 DPL,就产生一个“General protection”异常，因为中断

处理程序的特权不能低于引起中断的程序的特权。对于编程异常，则做进一步的安全检查：比较 CPL 与处于 IDT 中的门描述符的 DPL,如果 DPL 小于 CPL,就产生一个“General protection”异常。这最后一个检查可以避免用户应用程序访问特殊的陷阱门或中断门。

5. 检查是否发生了特权级的变化，也就是说，CPL 是否不同于所选择的段描述符的 DPL。

如果是，控制单元必须开始使用与新的特权级相关的栈。通过执行以下步骤来做到这点：

i. 读 tr 寄存器，以访问运行进程的 TSS 段。

ii. 用与新特权级相关的栈段和栈指针的正确值装载 ss 和 esp 寄存器。这些值可以在 TSS 中找到。

iii. 在新的栈中保存 ss 和 esp 以前的值，这些值定义了与旧特权级相关的栈的逻辑地址。

6. 如果故障已发生，用引起异常的指令地址装载 CS 和 eip 寄存器，从而使得这条指令能再次被执行。

7. 在栈中保存 eflags、CS 及 eip 的内容。

8. 如果异常产生了一个硬件出错码，则将它保存在栈中。

9. 装载 cs 和 eip 寄存器，其值分别是 IDT 表中第 i 项门描述符的段选择符和偏移量字段。

这些值给出了中断或者异常处理程序的第一条指令的逻辑地址。

控制单元所执行的最后一步就是跳转到中断或者异常处理程序。换句话说，处理完中断信号后，控制单元所执行的指令就是被选中处理程序的第一条指令。中断或异常被处理完后，相应的处理程序必须产生一条 iret 指令，把控制权转交给被中断的进程，这将迫使控制单元：

1. 用保存在栈中的值装载 CS、eip 或 eflags 寄存器。如果一个硬件出错码曾被压入栈中，并且在 eip 内容的上面，那么，执行 iret 指令前必须先弹出这个硬件出错码。

2. 检查处理程序的 CPL 是否等于 CS 中最低两位的值（这意味着被中断的进程与处理程序运行在同一特权级）。如果是，iret 终止执行；否则，转入下一步。

3. 从栈中装载 ss 和 esp 寄存器，因此，返回到与旧特权级相关的栈。

4. 检查 ds、es、fs 及 gs 段寄存器的内容，如果其中一个寄存器包含的选择符是一个段描述符，并且其 DPL 值小于 CPL，那么，清相应的段寄存器。控制单元这么做是为了禁止用户态的程序（CPL=3）利用内核以前所用的段寄存器

（DPL=0）。如果不清这些寄存器，怀有恶意的用户态程序就可能利用它们来访问内核地址空间。

2.5 中断向量表IDT的初始化

CPU 硬件为中断提供了一套机制，要是用这套中断机制，内核必须对相应的数据结构进行初始化。

Linux 内核在初始化阶段完成了对页式虚存管理之后，调用 trap_init 和 init_IRQ 两个函数进行中断机制的初始化。

2.5.1 trap_init

trap_init 主要是对一些系统保留的中断向量进行初始化。

```
882 void __init trap_init(void)
883 {
884         int i;
885
886 #ifdef CONFIG_EISA
887         void __iomem *p = early_ioremap(0x0FFFD9, 4);
888
889         if (readl(p) == 'E' + ('I'<<8) + ('S'<<16) + ('A'<<24))
890                 EISA_bus = 1;
891         early_iounmap(p, 4);
892 #endif
893
894         set_intr_gate(0, &divide_error);
895         set_intr_gate_ist(1, &debug, DEBUG_STACK);
896         set_intr_gate_ist(2, &nmi, NMI_STACK);
897         /* int3 can be called from all */
898         set_system_intr_gate_ist(3, &int3, DEBUG_STACK);
899         /* int4 can be called from all */
900         set_system_intr_gate(4, &overflow);
901         set_intr_gate(5, &bounds);
```

```
902         set_intr_gate(6, &invalid_op);
903         set_intr_gate(7, &device_not_available);
904 #ifdef CONFIG_X86_32
905         set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
906 #else
907         set_intr_gate_ist(8, &double_fault, DOUBLEFAULT_STACK);
908 #endif
909         set_intr_gate(9, &coprocessor_segment_overrun);
910         set_intr_gate(10, &invalid_TSS);
911         set_intr_gate(11, &segment_not_present);
912         set_intr_gate_ist(12, &stack_segment, STACKFAULT_STACK);
913         set_intr_gate(13, &general_protection);
914         set_intr_gate(14, &page_fault);
915         set_intr_gate(15, &spurious_interrupt_bug);
916         set_intr_gate(16, &coprocessor_error);
917         set_intr_gate(17, &alignment_check);
918 #ifdef CONFIG_X86_MCE
919         set_intr_gate_ist(18, &machine_check, MCE_STACK);
920 #endif
921         set_intr_gate(19, &simd_coprocessor_error);
```

894-921 行：设置中断向量表开头的 19 个陷阱门，这些是 CPU 保留用于异常处理的。

```
922
923         /* Reserve all the builtin and the syscall vector: */
924         for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
925                 set_bit(i, used_vectors);
926
927 #ifdef CONFIG_IA32_EMULATION
928         set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
929         set_bit(IA32_SYSCALL_VECTOR, used_vectors);
930 #endif
```

```
931
932 #ifdef CONFIG_X86_32
933         if (cpu_has_fxsr) {
934                 printk(KERN_INFO "Enabling fast FPU save and restore... ");
935                 set_in_cr4(X86_CR4_OSFXSR);
936                 printk("done.\n");
937         }
938         if (cpu_has_xmm) {
939                 printk(KERN_INFO
940                        "Enabling unmasked SIMD FPU exception support... ");
941                 set_in_cr4(X86_CR4_OSXMMEXCPT);
942                 printk("done.\n");
943         }
944         }
945         set_system_trap_gate(SYSCALL_VECTOR, &system_call);
```

系统调用向量初始化

```
946         set_bit(SYSCALL_VECTOR, used_vectors);
947 #endif
948
949         /*
950          * Should be a barrier for any external CPU state:
951          */
952         cpu_init();
953
954         x86_init.irqs.trap_init();
955 }
```

2.5.2 do_irq

arch/x86/kernel/irqinit.c

```
235 void __init native_init_IRQ(void)
```

```
236 {
237         int i;
238
239         /* Execute any quirks before the call gates are initialised: */
240         x86_init.irqs.pre_vector_init();
241
242         apic_intr_init();
243
244         /*
245          * Cover the whole vector space, no vector can escape
246          * us. (some of these will be overridden and become
247          * 'special' SMP interrupts)
248          */
249         for (i = FIRST_EXTERNAL_VECTOR; i < NR_VECTORS; i++) {
250                 /* IA32_SYSCALL_VECTOR could be used in trap_init already. */
251                 if (!test_bit(i, used_vectors))
252                         set_intr_gate(i, interrupt[i-FIRST_EXTERNAL_VECTOR]);
```

循环设置深入的中断向量表。

```
253         }
254
255         if (!acpi_ioapic)
256                 setup_irq(2, &irq2);
257
258 #ifdef CONFIG_X86_32
259         /*
260          * External FPU? Set up irq13 if so, for
261          * original braindamaged IBM FERR coupling.
262          */
263         if (boot_cpu_data.hard_math && !cpu_has_fpu)
264                 setup_irq(FPU_IRQ, &fpu_irq);
265
```

```
266         irq_ctx_init(smp_processor_id());
267 #endif
268 }
```

3. 中断的进入和返回

3.1 外设中断的进入和返回

3.1.1 中断的进入

IRQn 中的中断程序的地址开始存在 Interrupt[n]中，然后复制到 IDT 相应的表项中。

Interrunt[n]中存放下面两条汇编语言指令的地址：

push $n-256

jmp common_interrupt

arch/x86/kernel/entry_32.S

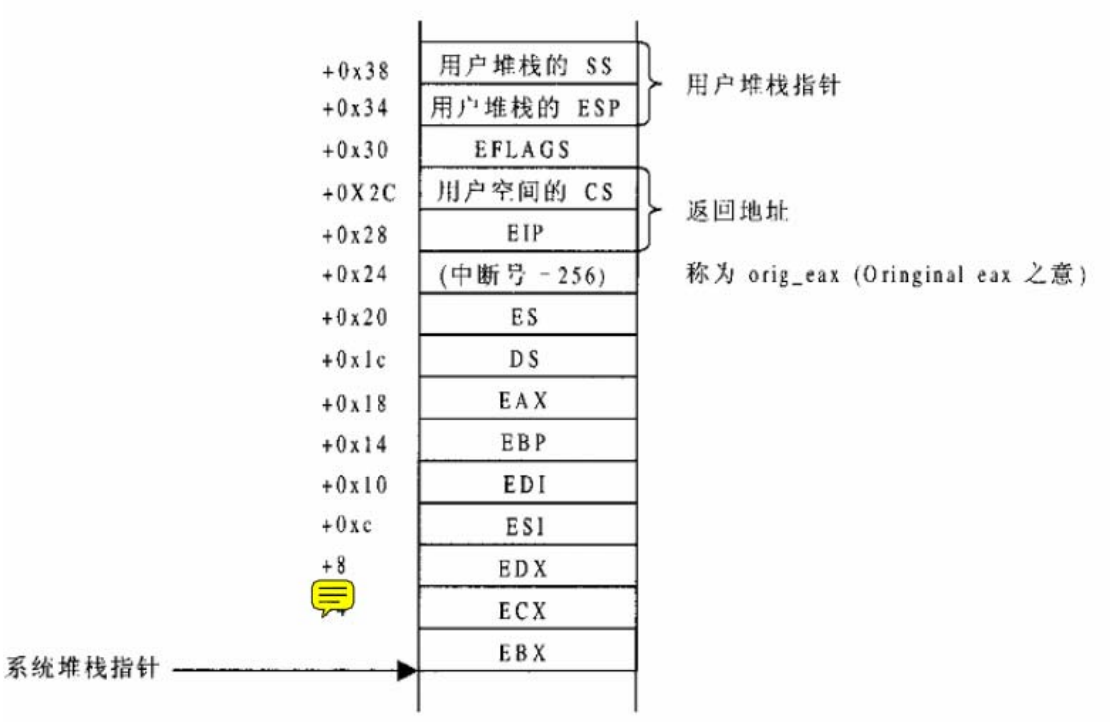
```
862 common_interrupt:
863         addl $-0x80,(%esp)    /* Adjust vector into the [-256,-1] range */
864         SAVE_ALL
865         TRACE_IRQS_OFF
866         movl %esp,%eax
867         call do_IRQ
868         jmp ret_from_intr
869 ENDPROC(common_interrupt)
```

```
194 .macro SAVE_ALL
195         cld
196         PUSH_GS
197         pushl %fs
198         CFI_ADJUST_CFA_OFFSET 4
```

```
199     /*CFI_REL_OFFSET fs, 0;*/
200     pushl %es
201     CFI_ADJUST_CFA_OFFSET 4
202     /*CFI_REL_OFFSET es, 0;*/
203     pushl %ds
204     CFI_ADJUST_CFA_OFFSET 4
205     /*CFI_REL_OFFSET ds, 0;*/
206     pushl %eax
207     CFI_ADJUST_CFA_OFFSET 4
208     CFI_REL_OFFSET eax, 0
209     pushl %ebp
210     CFI_ADJUST_CFA_OFFSET 4
211     CFI_REL_OFFSET ebp, 0
212     pushl %edi
213     CFI_ADJUST_CFA_OFFSET 4
214     CFI_REL_OFFSET edi, 0
215     pushl %esi
216     CFI_ADJUST_CFA_OFFSET 4
217     CFI_REL_OFFSET esi, 0
218     pushl %edx
219     CFI_ADJUST_CFA_OFFSET 4
220     CFI_REL_OFFSET edx, 0
221     pushl %ecx
222     CFI_ADJUST_CFA_OFFSET 4
223     CFI_REL_OFFSET ecx, 0
224     pushl %ebx
225     CFI_ADJUST_CFA_OFFSET 4
226     CFI_REL_OFFSET ebx, 0
227     movl $(__USER_DS), %edx
228     movl %edx, %ds
229     movl %edx, %es
```

```
230         movl $(__KERNEL_PERCPU), %edx
231         movl %edx, %fs
232         SET_KERNEL_GS %edx
233 .endm
```

这里可以看出，EFALGS 的内容并没有在 SAVE_ALL 中保存，这是因为 CUP 在进入中断服务时已经把它的内容连同返回地址一起压入堆栈了。至于原来的堆栈段寄存器 SS 和堆栈指针 ESP 的内容，则或者已被压入堆栈（如果更换过堆栈），或者继续使用而无需保存（不更换堆栈）。这样，在 SAVE_ALL 后，堆栈中的内容就变为：



save_all 之后，会调用 do_IRQ 函数：函数的调用参数是一个 pt_regs 数据结构。可以看出，该数据结构和上述的调用栈一致，之前 SAVE_ALL 所做的工作，包括 CUP 进入中断是自动做的，都是在为 do_IRQ 建立一个模拟的子程序调用环境，使得 do_IRQ 中既可以方便地知道进入中断前夕各个寄存器的内容，又可以在执行完毕后返回到 ret_from_intr。

arch/x86/include/asm/ptrace.h

```
21 struct pt_regs {
 22         long ebx;
 23         long ecx;
```

```
24          long edx;
25          long esi;
26          long edi;
27          long ebp;
28          long eax;
29          int  xds;
30          int  xes;
31          int  xfs;
32          int  xgs;
33          long orig_eax;
34          long eip;
35          int  xcs;
36          long eflags;
37          long esp;
38          int  xss;
39 };
```

```
221 /*
222  * do_IRQ handles all normal device IRQ's (the special
223  * SMP cross-CPU interrupts have their own specific
224  * handlers).
225  */
226 unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
227 {
228         struct pt_regs *old_regs = set_irq_regs(regs);
229
230         /* high bit used in ret_from_ code  */
231         unsigned vector = ~regs->orig_ax;
232         unsigned irq;
233
```

```
234         exit_idle();
235         irq_enter();
236
237         irq = __get_cpu_var(vector_irq)[vector];
238
239         if (!handle_irq(irq, regs)) {
240                 ack_APIC_irq();
241
242                 if (printk_ratelimit())
243                         pr_emerg("%s: %d.%d No irq handler for vector (irq %d)\n",
244                                 __func__, smp_processor_id(), vector, irq);
245         }
246
247         irq_exit();
248
249         set_irq_regs(old_regs);
250         return 1;
251 }
```

3.1.2 中断的返回

```
352 ret_from_intr:
353         GET_THREAD_INFO(%ebp)
354 check_userspace:
355         movl PT_EFLAGS(%esp), %eax    # mix EFLAGS and CS
356         movb PT_CS(%esp), %al
357         andl $(X86_EFLAGS_VM | SEGMENT_RPL_MASK), %eax
358         cmpl $USER_RPL, %eax
359           jb  resume_kernel               # not returning to v8086 or
userspace
360
```

```
361 ENTRY(resume_userspace)
362         LOCKDEP_SYS_EXIT
363          DISABLE_INTERRUPTS(CLBR_ANY)       # make sure we
don't miss an interrupt
364                                 # setting need_resched or sigpending
365                                 # between sampling and the iret
366         TRACE_IRQS_OFF
367         movl TI_flags(%ebp), %ecx
368         andl $_TIF_WORK_MASK, %ecx   # is there any work to be
done on
369                                 # int/exception return?
370         jne work_pending
371         jmp restore_all
372 END(ret_from_exception)
```

```
553 restore_all:
554         TRACE_IRQS_IRET
555 restore_all_notrace:
556         movl PT_EFLAGS(%esp), %eax    # mix EFLAGS, SS and CS
557          # Warning: PT_OLDSS(%esp) contains the wrong/random
values if we
558         # are returning to the kernel.
559         # See comments in process.c:copy_thread() for details.
560         movb PT_OLDSS(%esp), %ah
561         movb PT_CS(%esp), %al
562          andl $(X86_EFLAGS_VM | (SEGMENT_TI_MASK << 8) |
SEGMENT_RPL_MASK), %eax
563         cmpl $((SEGMENT_LDT << 8) | USER_RPL), %eax
564         CFI_REMEMBER_STATE
565         je ldt_ss                 # returning to user-space with LDT SS
566 restore_nocheck:
567         RESTORE_REGS 4                # skip orig_eax/error_code
```

```
568         CFI_ADJUST_CFA_OFFSET -4
569 irq_return:
570         INTERRUPT_RETURN
```

```
259 .macro RESTORE_REGS pop=0
260         RESTORE_INT_REGS
261 1:     popl %ds
262         CFI_ADJUST_CFA_OFFSET -4
263         /*CFI_RESTORE ds;*/
264 2:     popl %es
265         CFI_ADJUST_CFA_OFFSET -4
266         /*CFI_RESTORE es;*/
267 3:     popl %fs
268         CFI_ADJUST_CFA_OFFSET -4
269         /*CFI_RESTORE fs;*/
270         POP_GS \pop
271 .pushsection .fixup, "ax"
272 4:     movl $0, (%esp)
273         jmp 1b
274 5:     movl $0, (%esp)
275         jmp 2b
276 6:     movl $0, (%esp)
277         jmp 3b
278 .section __ex_table, "a"
279         .align 4
280         .long 1b, 4b
281         .long 2b, 5b
282         .long 3b, 6b
283 .popsection
284         POP_GS_EX
285 .endm
```

```
235 .macro RESTORE_INT_REGS
236         popl %ebx
237         CFI_ADJUST_CFA_OFFSET -4
238         CFI_RESTORE ebx
239         popl %ecx
240         CFI_ADJUST_CFA_OFFSET -4
241         CFI_RESTORE ecx
242         popl %edx
243         CFI_ADJUST_CFA_OFFSET -4
244         CFI_RESTORE edx
245         popl %esi
246         CFI_ADJUST_CFA_OFFSET -4
247         CFI_RESTORE esi
248         popl %edi
249         CFI_ADJUST_CFA_OFFSET -4
250         CFI_RESTORE edi
251         popl %ebp
252         CFI_ADJUST_CFA_OFFSET -4
253         CFI_RESTORE ebp
254         popl %eax
255         CFI_ADJUST_CFA_OFFSET -4
256         CFI_RESTORE eax
257 .endm
```

可以看到，RESTORE_ALL 是与进入内核时执行的宏操作 SAVE_ALL 遥相呼应的。570 行的 INTERRUPT_RETURN 也就是 Iret 使得 cpu 从中断返回。

3.2 异常的进入和返回

3.2.1 异常的进入

假设 handle_name 表示一个通用异常处理程序的名字。每一个异常处理程序都以下列的汇编指令开始：

```
1011 ENTRY(handle_name)
1012        RING0_INT_FRAME
1013        pushl $0                # no error code
1014        CFI_ADJUST_CFA_OFFSET 4
1015        pushl $do_handle_name
1016        CFI_ADJUST_CFA_OFFSET 4
1017        jmp error_code
1018        CFI_ENDPROC
```

如除 0 异常处理的如下：

```
1011 ENTRY(divide_error)
1012        RING0_INT_FRAME
1013        pushl $0                # no error code
1014        CFI_ADJUST_CFA_OFFSET 4
1015        pushl $do_divide_error
1016        CFI_ADJUST_CFA_OFFSET 4
1017        jmp error_code
1018        CFI_ENDPROC
1019 END(divide_error)
```

对于 error_code 看上去很熟悉，和 SAVE_ALL 功能基本相同，将函数可能用到的寄存器保存在栈中。

```
1272 error_code:
1273        /* the function address is in %gs's slot on the stack */
1274        pushl %fs
```

```
1275    CFI_ADJUST_CFA_OFFSET 4
1276    /*CFI_REL_OFFSET fs, 0*/
1277    pushl %es
1278    CFI_ADJUST_CFA_OFFSET 4
1279    /*CFI_REL_OFFSET es, 0*/
1280    pushl %ds
1281    CFI_ADJUST_CFA_OFFSET 4
1282    /*CFI_REL_OFFSET ds, 0*/
1283    pushl %eax
1284    CFI_ADJUST_CFA_OFFSET 4
1285    CFI_REL_OFFSET eax, 0
1286    pushl %ebp
1287    CFI_ADJUST_CFA_OFFSET 4
1288    CFI_REL_OFFSET ebp, 0
1289    pushl %edi
1290    CFI_ADJUST_CFA_OFFSET 4
1291    CFI_REL_OFFSET edi, 0
1292    pushl %esi
1293    CFI_ADJUST_CFA_OFFSET 4
1294    CFI_REL_OFFSET esi, 0
1295    pushl %edx
1296    CFI_ADJUST_CFA_OFFSET 4
1297    CFI_REL_OFFSET edx, 0
1298    pushl %ecx
1299    CFI_ADJUST_CFA_OFFSET 4
1300    CFI_REL_OFFSET ecx, 0
1301    pushl %ebx
1302    CFI_ADJUST_CFA_OFFSET 4
1303    CFI_REL_OFFSET ebx, 0
1304    cld
1305    movl $(__KERNEL_PERCPU), %ecx
```

```
1306        movl %ecx, %fs
1307        UNWIND_ESPFIX_STACK
1308        GS_TO_REG %ecx
1309        movl PT_GS(%esp), %edi          # get the function address
1310        movl PT_ORIG_EAX(%esp), %edx    # get the error code
1311        movl $-1, PT_ORIG_EAX(%esp) # no syscall to restart
```

将栈中 ORIG_EAX 处的的地方赋值-1，这个值用来区分 0x80 异常与其它异常。

```
1312        REG_TO_PTGS %ecx
1313        SET_KERNEL_GS %ecx
1314        movl $(__USER_DS), %ecx
1315        movl %ecx, %ds
1316        movl %ecx, %es
1317        TRACE_IRQS_OFF
1318        movl %esp,%eax                  # pt_regs pointer
1319        call *%edi //调用异常函数。
1320        jmp ret_from_exception
```

3.2.2 异常返回

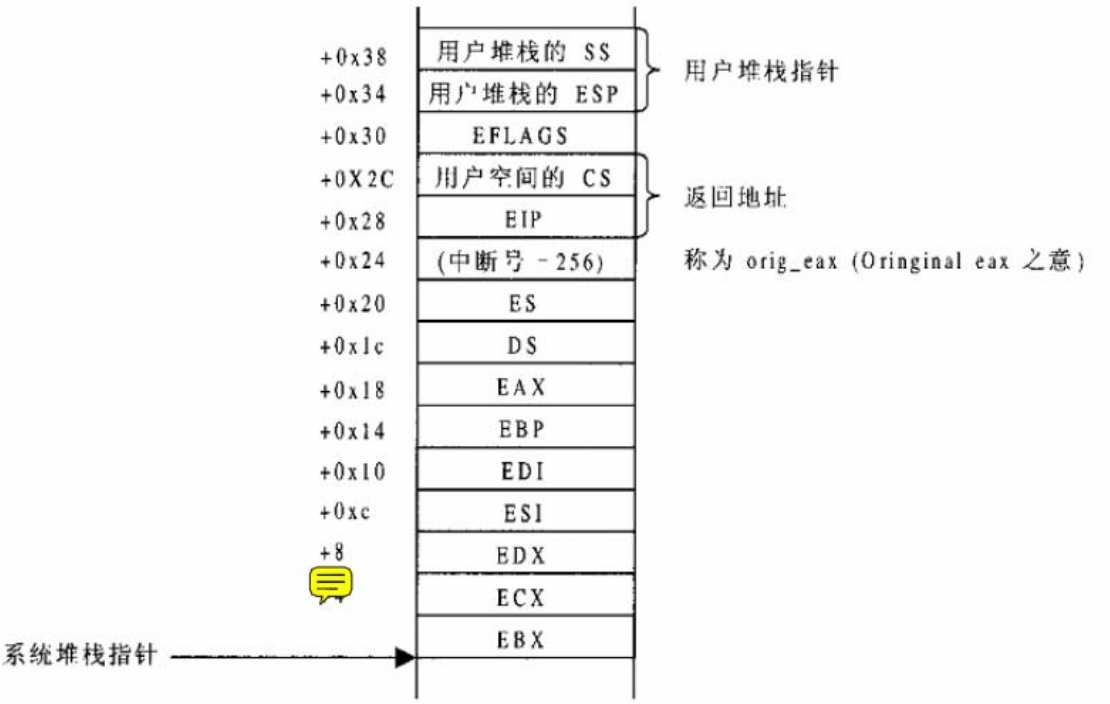
```
350 ret_from_exception:
 351        preempt_stop(CLBR_ANY)
 352 ret_from_intr:
 353        GET_THREAD_INFO(%ebp)
 354 check_userspace:
 355        movl PT_EFLAGS(%esp), %eax    # mix EFLAGS and CS
 356        movb PT_CS(%esp), %al
 357        andl $(X86_EFLAGS_VM | SEGMENT_RPL_MASK), %eax
 358        cmpl $USER_RPL, %eax
 359          jb  resume_kernel                 # not returning to v8086 or
userspace
```

```
360
361 ENTRY(resume_userspace)
362         LOCKDEP_SYS_EXIT
363          DISABLE_INTERRUPTS(CLBR_ANY)       # make sure we
don't miss an interrupt
364                                 # setting need_resched or sigpending
365                                 # between sampling and the iret
366         TRACE_IRQS_OFF
367         movl TI_flags(%ebp), %ecx
368         andl $_TIF_WORK_MASK, %ecx   # is there any work to be
done on
369                                 # int/exception return?
370         jne work_pending
371         jmp restore_all
372 END(ret_from_exception)
```

3.3 系统调用的进入和返回

外部中断是使 CPU 被动地、异常地进入系统空间的一种手段。系统调用是 CPU 主动的、同步地进入系统空间的手段，所谓“主动”，是指 CPU“自愿”的、事先计划好了的行为。而“同步”则是说，CPU 确切地知道在执行哪一条指令后就一定会进入系统空间。

Linux 内核在系统调用时是通过寄存器来传递参数的。为什么要用寄存器传递参数？当 CPU 穿过陷阱门，从用户空间进入系统空间时，由于运行级别的变动，要从用户堆栈切换到系统堆栈。如果在 INT 指令之前将参数压入堆栈，那是在用户堆栈中，而进入系统空间后就换成系统堆栈。虽然进入系统空间也可以访问用户空间堆栈，但比较费事。通过寄存器传递参数，则是个巧妙的安排。



从 SAVE_ALL 之后的系统堆栈看，%eax 持有调用号，不能用来传递参数。%ebp 是子程序调用过程的“帧”指针的，也不能用来传递参数，所以可以使用的参数也就只有 edi,esi,edx,ecx,ebx 最后 5 个寄存器了。所以，在系统调用过程中独立传递的参数不能超过 5 个。

3.3.1 系统调用的进入和返回

```
529 ENTRY(system_call)
530     RINGO_INT_FRAME             # can't unwind into user space anyway
531     pushl %eax                  # save orig_eax
532     CFI_ADJUST_CFA_OFFSET 4
533     SAVE_ALL
534     GET_THREAD_INFO(%ebp)
535                                 # system call tracing in operation / emulation
536     testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
537     jnz syscall_trace_entry
538     cmpl $(nr_syscalls), %eax
539     jae syscall_badsys
```

判断 PT_TRACESYS 是否置位，如果置位，则跳转到 syscall_trace_entry。

```
540 syscall_call:
541         call *sys_call_table(,%eax,4)
542         movl %eax,PT_EAX(%esp)          # store the return value
```

调用内核中对应的函数，同事将返回值保存到堆栈中。

```
543 syscall_exit:
544         LOCKDEP_SYS_EXIT
545          DISABLE_INTERRUPTS(CLBR_ANY)       # make sure we
don't miss an interrupt
546                                 # setting need_resched or sigpending
547                                 # between sampling and the iret
548         TRACE_IRQS_OFF
549         movl TI_flags(%ebp), %ecx
550         testl $_TIF_ALLWORK_MASK, %ecx    # current->work
551         jne syscall_exit_work
552
553 restore_all:
554         TRACE_IRQS_IRET
555 restore_all_notrace:
556         movl PT_EFLAGS(%esp), %eax    # mix EFLAGS, SS and CS
557          # Warning: PT_OLDSS(%esp) contains the wrong/random
values if we
558         # are returning to the kernel.
559         # See comments in process.c:copy_thread() for details.
560         movb PT_OLDSS(%esp), %ah
561         movb PT_CS(%esp), %al
562          andl  $(X86_EFLAGS_VM | (SEGMENT_TI_MASK << 8) |
SEGMENT_RPL_MASK), %eax
563         cmpl $((SEGMENT_LDT << 8) | USER_RPL), %eax
564         CFI_REMEMBER_STATE
565         je ldt_ss                 # returning to user-space with LDT SS
566 restore_nocheck:
```

```
567         RESTORE_REGS 4              # skip orig_eax/error_code
 568         CFI_ADJUST_CFA_OFFSET -4
 569 irq_return:
 570         INTERRUPT_RETURN
```

返回到调用空间。

4. 软中断

中断服务例程中，有些由内核执行的任务是不紧急的：在必要的情况下可以延迟一段时间。一个中断处理程序的几个中断服务例程之间是串行执行的，并且通常在一个中断的处理程序结束之前，不应该在出现这个中断，相反，可延长中断可以在开中断的情况下执行。把可延迟中断从中断处理程序中抽出来，有助于使内核保持较短的响应时间。

Linux 2.6 内核通过软中断和 tasklet 来解决这种问题。

软中断的分配是静态的（即在编译时定义），而 tasklet 的分配和初始化可以在运行时进行（例如：安装一个内核模块时）。软中断（即便是同一种类型的软中断）可以并发地运行在多个 CPU 上。因此，软中断是可重入函数而且必须明确地使用自旋锁保护其数据结构。tasklet 不必担心这些问题，因为内核对 tasklet 的执行进行了更加严格的控制。相同类型的 tasklet 总是被串行地执行，换句话说就是：不能在两个 CPU 上同时运行相同类型的 tasklet。但是，

类型不同的 tasklet 可以在几个 CPU 上并发执行。tasklet 的串行化使 tasklet 函数不必是可重入的，因此简化了设备驱动程序开发者的工作。

一般而言，在可延迟函数上可以执行四种操作：

初始化（initialization）

定义一个新的可延迟函数；这个操作通常在内核自身初始化或加载模块时进行。

激活（activation）

标记一个可延迟函数为“挂起”（在可延迟函数的下一轮调度中执行）。激活可以在任何时候

进行（即使正在处理中断）。

屏蔽（masking）

有选择地屏蔽一个可延迟函数，这样，即使它被激活，内核也不执行它。

执行（execution）

执行一个挂起的可延迟函数和同类型的其他所有挂起的可延迟函数；执行是在特定的时间进行的。

4.1 软中断

Linux 2.6 使用有限个软中断。其实，Linux 更倾向于用 tasklet，因为大多数场合 tasklet 是足够用的，且更容易编写，因为 tasklet 不必是可重入的。如下表所示，目前只定义了六种软中断。

软中断	下标	（优先级）
HI_SOFTIRQ	0	处理高优先级的 tasklet
TIMER_SOFTIRQ	1	和时钟中断相关的 tasklet
NET_TX_SOFTIRQ	2	把数据包传送到网卡
NET_RX_SOFTIRQ	3	从网卡接收数据包
SCSI_SOFTIRQ	4	SCSI
TASKLET_SOFTIRQ	5	处理常规 tasklet

一个软中断的下标决定了它的优先级：低下标意味着高优先级，因为软中断函数将从下标 0 开始执行。

4.1.1 软中断所使用的数据结构

Include/linux/interrupt.h

```
341 enum
342 {
343         HI_SOFTIRQ=0,
344         TIMER_SOFTIRQ,
345         NET_TX_SOFTIRQ,
346         NET_RX_SOFTIRQ,
347         BLOCK_SOFTIRQ,
348         BLOCK_IOPOLL_SOFTIRQ,
349         TASKLET_SOFTIRQ,
350         SCHED_SOFTIRQ,
351         HRTIMER_SOFTIRQ,
352         RCU_SOFTIRQ,      /* Preferable RCU should always be the last softirq */
```

```
353
354         NR_SOFTIRQS
355 };
```

```
366 struct softirq_action
367 {
368         void    (*action)(struct softirq_action *);
369 };
```

Kernel/softirq.c

```
55      static  struct  softirq_action  softirq_vec[NR_SOFTIRQS]
__cacheline_aligned_in_smp;
```

表示软中断的主要数据结构是 softirq_vec 数组。一个软中断的优先级是相应的 softirq_action 元素在数组内的下标。如上表所示，只有数组的前六个元素被有效地使用。softirq_action 数据结构包括字段阿 action 指向软中断函数。

4.1.2 软中断的处理

4.1.2.1 软中断初始化softirq_init

```
674 void __init softirq_init(void)
675 {
676         int cpu;
677
678         for_each_possible_cpu(cpu) {
679                 int i;
680
681                 per_cpu(tasklet_vec, cpu).tail =
682                         &per_cpu(tasklet_vec, cpu).head;
683                 per_cpu(tasklet_hi_vec, cpu).tail =
684                         &per_cpu(tasklet_hi_vec, cpu).head;
685                 for (i = 0; i < NR_SOFTIRQS; i++)
686                         INIT_LIST_HEAD(&per_cpu(softirq_work_list[i], cpu));
```

```
687         }
688
689         register_hotcpu_notifier(&remote_softirq_cpu_notifier);
690
691         open_softirq(TASKLET_SOFTIRQ, tasklet_action);
692         open_softirq(HI_SOFTIRQ, tasklet_hi_action);
693 }
```

```
343 void open_softirq(int nr, void (*action)(struct softirq_action *))
344 {
345         softirq_vec[nr].action = action;
346 }
```

open_softirq()函数处理软中断的初始化。它使用二个参数：软中断下标、指向要执行的软中断函数的指针。open_softirq()限制自己初始化 softirq_vec 数组中适当的元素。

4.1.2.2 激活软中断 raise_softirq()

```
7 typedef struct {
 8         unsigned int __softirq_pending;
 9         unsigned int __nmi_count;      /* arch dependent */
10         unsigned int irq0_irqs;
11 #ifdef CONFIG_X86_LOCAL_APIC
12         unsigned int apic_timer_irqs;      /* arch dependent */
13         unsigned int irq_spurious_count;
14 #endif
15         unsigned int x86_platform_ipis; /* arch dependent */
16         unsigned int apic_perf_irqs;
17         unsigned int apic_pending_irqs;
18 #ifdef CONFIG_SMP
19         unsigned int irq_resched_count;
```

```
20        unsigned int irq_call_count;
21        unsigned int irq_tlb_count;
22 #endif
23 #ifdef CONFIG_X86_THERMAL_VECTOR
24        unsigned int irq_thermal_count;
25 #endif
26 #ifdef CONFIG_X86_MCE_THRESHOLD
27        unsigned int irq_threshold_count;
28 #endif
29 } ____cacheline_aligned irq_cpustat_t;
```

```
314 /*
315    * This function must run with irqs disabled!
316    */
317 inline void raise_softirq_irqoff(unsigned int nr)
318 {
319        __raise_softirq_irqoff(nr);
320
321        /*
322         * If we're in an interrupt or softirq, we're done
323         * (this also catches softirq-disabled code). We will
324         * actually run the softirq once we return from
325         * the irq or softirq.
326         *
327         * Otherwise we wake up ksoftirqd to make sure we
328         * schedule the softirq soon.
329         */
330        if (!in_interrupt())
331                wakeup_softirqd();
332 }
```

```
375 #define __raise_softirq_irqoff(nr) do { or_softirq_pending(1UL << (nr)); } while (0)
```

```
45 #define or_softirq_pending(x)    percpu_or(irq_stat.__softirq_pending, (x))
```

```
19 #ifndef __ARCH_IRQ_STAT
20 extern irq_cpustat_t irq_stat[];          /* defined in asm/hardirq.h */
21 #define __IRQ_STAT(cpu, member) (irq_stat[cpu].member)
22 #endif
```

4.1.2.3 执行软中断do_softirq

```
253 asmlinkage void do_softirq(void)
254 {
255         __u32 pending;
256         unsigned long flags;
257
258         if (in_interrupt())
259                 return;
```

如果是在中断中，则返回。

```
260
261         local_irq_save(flags);
262
263         pending = local_softirq_pending();
264
265         if (pending)
266                 __do_softirq();
```

如果有软中断，则调用__do_softirq。

```
267
268         local_irq_restore(flags);
```

```
269 }
```

```
191 asmlinkage void __do_softirq(void)
192 {
193         struct softirq_action *h;
194         __u32 pending;
195         int max_restart = MAX_SOFTIRQ_RESTART;
196         int cpu;
197
198         pending = local_softirq_pending();
199         account_system_vtime(current);
200
201                                             __local_bh_disable((unsigned
long)__builtin_return_address(0));
202         lockdep_softirq_enter();
203
204         cpu = smp_processor_id();
205 restart:
206         /* Reset the pending bitmask before enabling irqs */
207         set_softirq_pending(0);
208
209         local_irq_enable();
210
211         h = softirq_vec;
212
213         do {
214                 if (pending & 1) {
215                         int prev_count = preempt_count();
216                         kstat_incr_softirqs_this_cpu(h - softirq_vec);
217
218                         trace_softirq_entry(h, softirq_vec);
```

```
219                 h->action(h);
```

指向软中断。

```
220                 trace_softirq_exit(h, softirq_vec);
221                 if (unlikely(prev_count != preempt_count())) {
222                         printk(KERN_ERR "huh, entered softirq %td %s
%p"
223                                "with preempt_count %08x,"
224                                " exited with %08x?\n", h - softirq_vec,
225                                softirq_to_name[h - softirq_vec],
226                                h->action, prev_count, preempt_count());
227                         preempt_count() = prev_count;
228                 }
229
230                 rcu_bh_qs(cpu);
231             }
232             h++;
233             pending >>= 1;
234         } while (pending);
```

213-234 循环执行 CPU 上的软中断。

```
235
236         local_irq_disable();
237
238         pending = local_softirq_pending();
239         if (pending && --max_restart)
240                 goto restart;
```

如果在执行软中断的过程中有软中断发生，则在上述 do-while 循环执行的次数不超过 max_restart 时，跳转到 restart。

```
241
242         if (pending)
243                 wakeup_softirqd();
```

否则，唤醒内核线程 ksoftirqd 执行软中断。

```
244
245         lockdep_softirq_exit();
246
247         account_system_vtime(current);
248         _local_bh_enable();
249 }
```

4.1.2.4 ksoftirq 内核线程

```
695 static int run_ksoftirqd(void * __bind_cpu)
696 {
697         set_current_state(TASK_INTERRUPTIBLE);
698
699         while (!kthread_should_stop()) {
700                 preempt_disable();
701                 if (!local_softirq_pending()) {
702                         preempt_enable_no_resched();
703                         schedule();
704                         preempt_disable();
705                 }
706
707                 __set_current_state(TASK_RUNNING);
708
709                 while (local_softirq_pending()) {
710                         /* Preempt disable stops cpu going offline.
711                            If already offline, we'll be on wrong CPU:
712                            don't process */
713                         if (cpu_is_offline((long)__bind_cpu))
714                                 goto wait_to_die;
715                         do_softirq();
716                         preempt_enable_no_resched();
```

```
717                 cond_resched();
718                 preempt_disable();
719                 rcu_sched_qs((long)__bind_cpu);
720             }
```

While 循环中运行 so_softirq 函数。

```
721             preempt_enable();
722             set_current_state(TASK_INTERRUPTIBLE);
723         }
724         __set_current_state(TASK_RUNNING);
725         return 0;
726
727 wait_to_die:
728         preempt_enable();
729         /* Wait for kthread_stop */
730         set_current_state(TASK_INTERRUPTIBLE);
731         while (!kthread_should_stop()) {
732             schedule();
733             set_current_state(TASK_INTERRUPTIBLE);
734         }
735         __set_current_state(TASK_RUNNING);
736         return 0;
737 }
```

4.2 tasklet

tasklet 是 I/O 驱动程序中实现可延迟函数的首选方法。

如前所述，tasklet 建立在两个叫做 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 的软中断之上。几个 tasklet 可以与同一个软中断相关联，每个 tasklet 执行自己的函数。两个软中断之间没有真正的区别，只不过 do_softirq()先执行 HI_SOFTIRQ 的 tasklet，后执行 TASKLET_SOFTIRQ 的 tasklet。

tasklet 和高优先级的 tasklet 分别存放在 tasklet_vec 和 tasklet_hi_vec 数组中。二者都包含类型为 tasklet_head 的 NR_CPUS 个元素，每个元素都由一个指向 tasklet 描述符链表的指针组成。

4.2.1 数据结构

tasklet 描述符是一个 tasklet_struct 类型的数据结构。

include/linux/interrupt.h

```
420 struct tasklet_struct
421 {
422         struct tasklet_struct *next;
423         unsigned long state;
424         atomic_t count;
425         void (*func)(unsigned long);
426         unsigned long data;
427 };
```

func 指向要执行的函数。

next 指向下一个 tasklet_struct 实例。

state：表示任务当前状态。

```
348 /*
349    * Tasklets
350    */
351 struct tasklet_head
352 {
353         struct tasklet_struct *head;
354         struct tasklet_struct **tail;
355 };
356
357 static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
358 static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);
```

tasklet 和高优先级的 tasklet 分别存放在 tasklet_vec 和 tasklet_hi_vec 数组中。二者都包含类型为 tasklet_head 的 NR_CPUS 个元素，每个元素都由一个指向 tasklet 描述符链表的指针组成。

4.2.2 初始化tasklet_init

```
470 void tasklet_init(struct tasklet_struct *t,
471                     void (*func)(unsigned long), unsigned long data)
472 {
473         t->next = NULL;
474         t->state = 0;
475         atomic_set(&t->count, 0);
476         t->func = func;
477         t->data = data;
478 }
```

初始化 tasklet_struct 实例。

4.2.3 tasklet_schedule注册

include/linux/interrup.h

```
464 extern void __tasklet_schedule(struct tasklet_struct *t);
465
466 static inline void tasklet_schedule(struct tasklet_struct *t)
467 {
468         if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
469                 __tasklet_schedule(t);
470 }
```

```
360 void __tasklet_schedule(struct tasklet_struct *t)
361 {
362         unsigned long flags;
```

```
363
364         local_irq_save(flags);
365         t->next = NULL;
366         *__get_cpu_var(tasklet_vec).tail = t;
367         __get_cpu_var(tasklet_vec).tail = &(t->next);
368         raise_softirq_irqoff(TASKLET_SOFTIRQ);
369         local_irq_restore(flags);
370 }
371
372 EXPORT_SYMBOL(__tasklet_schedule);
```

4.2.4 执行tasklet

在软中断的初始化函数 softirq_init 中，有如下代码

```
691         open_softirq(TASKLET_SOFTIRQ, tasklet_action);
692         open_softirq(HI_SOFTIRQ, tasklet_hi_action);
```

所以 tasklet 的执行是调用 tasklet_action 和 tasklet_hi_action 函数的。这里只看 tasklet_action 函数。

```
399 static void tasklet_action(struct softirq_action *a)
400 {
401         struct tasklet_struct *list;
402
403         local_irq_disable();
404         list = __get_cpu_var(tasklet_vec).head;
405         __get_cpu_var(tasklet_vec).head = NULL;
406                                         __get_cpu_var(tasklet_vec).tail         =
&__get_cpu_var(tasklet_vec).head;
407         local_irq_enable();
408
409         while (list) {
410                 struct tasklet_struct *t = list;
```

```
411
412             list = list->next;
413
414             if (tasklet_trylock(t)) {
415                     if (!atomic_read(&t->count)) {
416                             if (!test_and_clear_bit(TASKLET_STATE_SCHED,
&t->state))
417                                     BUG();
418                             t->func(t->data);
```

执行 tasklet 的函数。

```
419                             tasklet_unlock(t);
420                             continue;
421                     }
422                     tasklet_unlock(t);
423             }
424
425             local_irq_disable();
426             t->next = NULL;
427             *__get_cpu_var(tasklet_vec).tail = t;
428             __get_cpu_var(tasklet_vec).tail = &(t->next);
429             __raise_softirq_irqoff(TASKLET_SOFTIRQ);
430             local_irq_enable();
431     }
432 }
```

5. 参考书籍

[1] linux 内核 2.6.34 源码

[2] 深入了解 linux 内核 （第三版）

[3] linxu 内核源代码情景分析 毛德超

[4] 深入 linxu 内核架构
[5] http://blog.csdn.net/yunsongice(网络资源)