第三章　中断机制

中断控制是计算机发展中一种重要的技术。最初它是为克服对 I/O 接口控制采用程序查询所带来的处理器低效率而产生的。中断控制的主要优点是只有在 I/O 需要服务时才能得到处理器的响应，而不需要处理器不断地进行查询。由此，最初的中断全部是对外部设备而言的，即称为外部中断（或硬件中断）。

但随着计算机系统结构的不断改进以及应用技术的日益提高，中断的适用范围也随之扩大，出现了所谓的内部中断（或叫异常），它是为解决机器运行时所出现的某些随机事件及编程方便而出现的。因而形成了一个完整的中断系统。

本章主要讨论在 Intel i386 保护模式下中断机制在 Linux 中的实现。如果你曾有过在 DOS 实模式下编写中断程序的经历，那么，你会发现中断在 Linux 中的实现较为复杂，这是由保护模式的复杂性引起的，因此，必须首先了解硬件机制对中断的支持。不过，不管在实模式下还是保护模式下，有关中断实现的基本原理是完全相同的。

3.1　中断基本知识

大多数读者可能对 16 位实地址模式下的中断机制有所了解，例如中断向量、外部 I/O 中断以及异常，这些内容在 32 位的保护模式下依然有效。两种模式之间最本质的差别就是在保护模式引入的中断描述符表。

3.1.1　中断向量

Intel x86 系列微机共支持 256 种向量中断，为使处理器较容易地识别每种中断源，将它们从 0～256 编号，即赋予一个中断类型码 *n*，Intel 把这个 8 位的无符号整数叫做一个向量,因此,也叫中断向量。所有 256 种中断可分为两大类:异常和中断。异常又分为故障(Fault)和陷阱（Trap），它们的共同特点是既不使用中断控制器，又不能被屏蔽。中断又分为外部可屏蔽中断（INTR）和外部非屏蔽中断（NMI），所有 I/O 设备产生的中断请求（IRQ）均引起屏蔽中断，而紧急的事件（如硬件故障）引起的故障产生非屏蔽中断。

非屏蔽中断的向量和异常的向量是固定的，而屏蔽中断的向量可以通过对中断控制器的编程来改变。Linux 对 256 个向量的分配如下。

- 从 0~31 的向量对应于异常和非屏蔽中断。
- 从 32~47 的向量（即由 I/O 设备引起的中断）分配给屏蔽中断。
- 剩余的从 48~255 的向量用来标识软中断。Linux 只用了其中的一个（即 128 或 0x80

向量）用来实现系统调用。当用户态下的进程执行一条 int 0x80 汇编指令时，CPU 就切换到内核态，并开始执行 system_call（　）内核函数。

3.1.2　外设可屏蔽中断

Intel x86 通过两片中断控制器 8259A 来响应 15 个外中断源，每个 8259A 可管理 8 个中断源。第 1 级（称主片）的第 2 个中断请求输入端，与第 2 级 8259A（称从片）的中断输出端 INT 相连，如图 3.1 所示。我们把与中断控制器相连的每条线叫做中断线，要使用中断线，就得进行中断线的申请，就是 IRQ（Interrupt ReQuirement ），我们也常把申请一条中断线称为申请一个 IRQ 或者是申请一个中断号。IRQ 线是从 0 开始顺序编号的，因此，第一条 IRQ 线通常表示成 IRQ0。IRQn 的缺省向量是 *n*+32；如前所述，IRQ 和向量之间的映射可以通过中断控制器端口来修改。

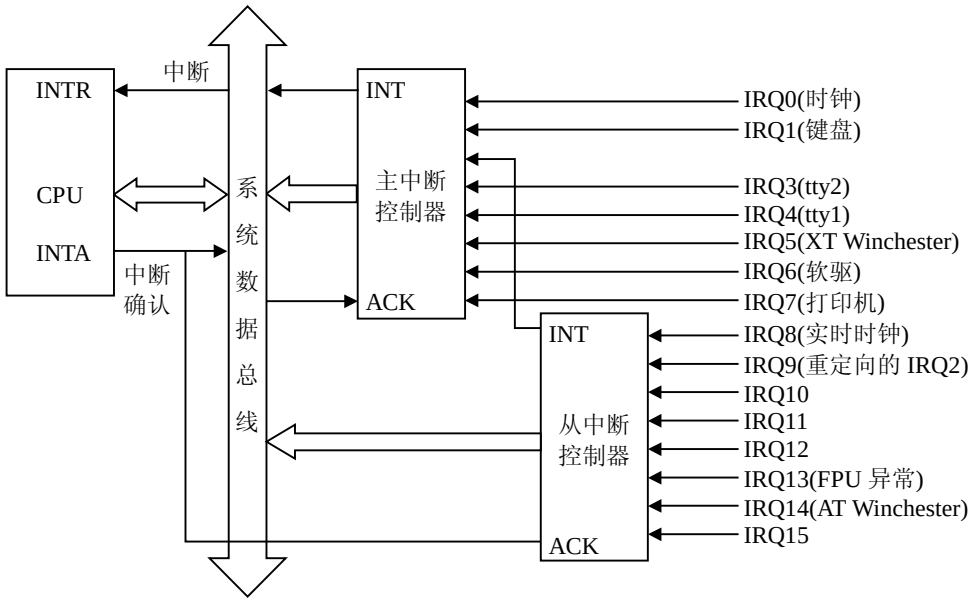


图 3.1　级连的 8259A 的中断机构

并不是每个设备都可以向中断线上发中断信号，只有对某一条确定的中断线拥有了控制权，才可以向这条中断线上发送信号。由于计算机的外部设备越来越多，所以 15 条中断线已经不够用了。中断线是非常宝贵的资源，只有当设备需要中断的时候才申请占用一个 IRQ，或者是在申请 IRQ 时采用共享中断的方式，这样可以让更多的设备使用中断。

中断控制器 8259A 执行如下操作。

（1）监视中断线，检查产生的中断请求（IRQ）信号。

（2）如果在中断线上产生了一个中断请求信号。

a. 把接受到的 IRQ 信号转换成一个对应的向量。

b. 把这个向量存放在中断控制器的一个 I/O 端口，从而允许 CPU 通过数据总线读此向量。

c. 把产生的信号发送到 CPU 的 INTR 引脚——即发出一个中断。

d. 等待，直到 CPU 确认这个中断信号，然后把它写进可编程中断控制器（PIC）的一个 I/O 端口；此时，清 INTR 线。

（3）返回到第一步。

对于外部 I/O 请求的屏蔽可分为两种情况，一种是从 CPU 的角度，也就是清除 eflag 的中断标志位（IF），当 IF=0 时，禁止任何外部 I / O 的中断请求，即关中断；一种是从中断控制器的角度，因为中断控制器中有一个 8 位的中断屏蔽寄存器（IMR），每位对应 8259A 中的一条中断线，如果要禁用某条中断线，则把 IRM 相应的位置 1，要启用，则置 0。

3.1.3　异常及非屏蔽中断

异常就是 CPU 内部出现的中断，也就是说，在 CPU 执行特定指令时出现的非法情况。非屏蔽中断就是计算机内部硬件出错时引起的异常情况。从图 3.1 可以看出，二者与外部 I/O 接口没有任何关系。Intel 把非屏蔽中断作为异常的一种来处理，因此，后面所提到的异常也包括了非屏蔽中断。在 CPU 执行一个异常处理程序时，就不再为其他异常或可屏蔽中断请求服务，也就是说，当某个异常被响应后，CPU 清除 eflag 的中 IF 位，禁止任何可屏蔽中断。但如果又有异常产生，则由 CPU 锁存（CPU 具有缓冲异常的能力），待这个异常处理完后，才响应被锁存的异常。我们这里讨论的异常中断向量在 0～31 之间，不包括系统调用（中断向量为 0x80）。

Intel x86 处理器发布了大约 20 种异常（具体数字与处理器模式有关）。Linux 内核必须为每种异常提供一个专门的异常处理程序。这里特别说明的是，在某些异常处理程序开始执行之前，CPU 控制单元会产生一个硬件错误码，内核先把这个错误码压入内核栈中。

在表 3.1 中给出了 Pentium 模型中异常的向量、名字、类型及简单描述。更多的信息可以在 Intel 的技术文挡中找到。

表 3.1　　异常的简单描述

向量	异常名	类别	描述
0	除法出错	故障	被 0 除
1	调试	故障 / 陷阱	当对一个程序进行逐步调试时
2	非屏蔽中断（NMI）	为不可屏蔽中断保留	
3	断点	陷阱	由 int3（断点指令）指令引起
4	溢出	陷阱	当 into（check for overflow）指令被执行
5	边界检查	故障	当 bound 指令被执行
6	非法操作码	故障	当 CPU 检查到一个无效的操作码
7	设备不可用	故障	随着设置 cr0 的 TS 标志，ESCAPE 或 MMX 指令被执行
8	双重故障	故障	处理器不能串行处理异常而引起的

续表

向量	异常名	类别	描述
9	协处理器段越界	故障	因外部的数学协处理器引起的问题（仅用在 80386）
10	无效 TSS	故障	要切换到的进程具有无效的 TSS
11	段不存在	故障	引用一个不存在的内存段
12	栈段异常	故障	试图超过栈段界限，或由 ss 标识的段不在内存
13	通用保护	故障	违反了 Intelx86 保护模式下的一个保护规则
14	页异常	故障	寻址的页不在内存，或违反了一种分页保护机制
15	Intel 保留	/	保留
16	浮点出错	故障	浮点单元用信号通知一个错误情形，如溢出
17	对齐检查	故障	操作数的地址没有被正确地排列

18～31 由 Intel 保留，为将来的扩充用。

另外，如表 3.2 所示，每个异常都由专门的异常处理程序来处理（参见本章后面“异常处理“部分），它们通常把一个 UNIX 信号发送到引起异常的进程。

表 3.2　　由异常处理程序发送的信号

向量	异常名	出错码	异常处理程序	信号
0	除法出错	/	divide_error（ ）	SIGFPE
1	调试	/	debug（ ）	SIGTRAP
2	非屏蔽中断（NMI）	/	nmi（ ）	None
3	断点	/	int3（ ）	SIGTRAP
4	溢出	/	overflow（ ）	SIGSEGV
5	边界检查	/	bounds（ ）	SIGSEGV
6	非法操作码	/	invalid_op（ ）	SIGILL
7	设备不可用	/	device_not_available（ ）	SIGSEGV
8	双重故障	有	double_fault（ ）	SIGSEGV
9	协处理器段越界	/	coprocessor_segment_overrun（ ）	SIGFPE
10	无效 TSS	有	invalid_tss（ ）	SIGSEGV
11	段不存在	有	segment_not_present（ ）	SIGBUS
12	栈段异常	有	stack_segment（ ）	SIGBUS
13	通用保护	有	general_protection（ ）	SIGSEGV
14	页异常	有	page_fault（ ）	SIGSEGV

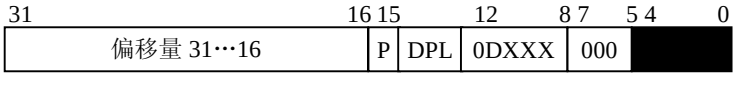
15	Intel 保留	/	None	None

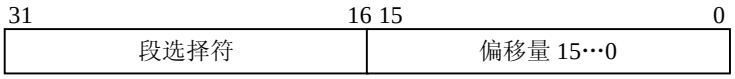
续表

向量	异常名	出错码	异常处理程序	信号
16	浮点出错	/	coprocessor_error（ ）	SIGFPE
17	对齐检查	/	alignment_check（ ）	SIGSEGV

3.1.4　中断描述符表

在实地址模式中，CPU 把内存中从 0 开始的 1K 字节作为一个中断向量表。表中的每个表项占 4 个字节，由两个字节的段地址和两个字节的偏移量组成，这样构成的地址便是相应中断处理程序的入口地址。但是，在实模式下，由 4 字节的表项构成的中断向量表显然满足不了要求。这是因为，①除了两个字节的段描述符，偏移量必用 4 字节来表示；②要有反映模式切换的信息。因此，在实模式下，中断向量表中的表项由 8 个字节组成，如图 3.2 所示，中断向量表也改叫做中断描述符表 IDT（Interrupt Descriptor Table）。其中的每个表项叫做一个门描述符（Gate Descriptor），“门”的含义是当中断发生时必须先通过这些门，然后才能进入相应的处理程序。





DPL	段描述符的特权级
偏移量	入口函数地址的偏移量
P	段是否在内存中的标志
段选择符	入口函数所处代码段的选择符
D	标志位，1=32 位，0=16 位
XXX	3 位门类型码

图 3.2　门描述符的一般格式

其中类型占 3 位，表示门描述符的类型，这些描述符如下。

1．任务门（Task gate）

其类型码为 101，门中包含了一个进程的 TSS 段选择符，但偏移量部分没有使用，因为 TSS 本身是作为一个段来对待的，因此，任务门不包含某一个入口函数的地址。TSS 是 Intel 所提供的任务切换机制，但是 Linux 并没有采用任务门来进行任务切换（参见第五章的任务切换）。

2．中断门（Interrupt gate）

其类型码为 110，中断门包含了一个中断或异常处理程序所在段的选择符和段内偏移量。当控制权通过中断门进入中断处理程序时，处理器清 IF 标志，即关中断，以避免嵌套中断的

发生。中断门中的 DPL（Descriptor Privilege Level）为 0，因此，用户态的进程不能访问 Intel 的中断门。所有的中断处理程序都由中断门激活，并全部限制在内核态。

3．陷阱门（Trap gate）

其类型码为 111，与中断门类似，其唯一的区别是，控制权通过陷阱门进入处理程序时维持 IF 标志位不变，也就是说，不关中断。

4．系统门（System gate）

这是 Linux 内核特别设置的，用来让用户态的进程访问 Intel 的陷阱门，因此，门描述符的 DPL 为 3。通过系统门来激活 4 个 Linux 异常处理程序，它们的向量是 3、4、5 及 128，也就是说，在用户态下，可以使用 int3、into、bound 及 int0x80 四条汇编指令。

最后，在保护模式下，中断描述符表在内存的位置不再限于从地址 0 开始的地方，而是可以放在内存的任何地方。为此，CPU 中增设了一个中断描述符表寄存器 IDTR，用来存放中断描述符表在内存的起始地址。中断描述符表寄存器 IDTR 是一个 48 位的寄存器，其低 16 位保存中断描述符表的大小，高 32 位保存 IDT 的基址，如图 3.3 所示。

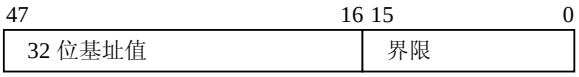


图 3.3 中断描述符表寄存器 IDTR

3.1.5 相关汇编指令

为了有助于读者对中断实现过程的理解，下面介绍几条相关的汇编指令。

1．调用过程指令 CALL

指令格式：CALL 过程名

说明：i386 在取出 CALL 指令之后及执行 CALL 指令之前，使指令指针寄存器 EIP 指向紧接 CALL 指令的下一条指令。CALL 指令先将 EIP 值压入栈内，再进行控制转移。当遇到 RET 指令时，栈内信息可使控制权直接回到 CALL 的下一条指令

2．调用中断过程指令 INT

指令格式：INT 中断向量

说明：EFLAG、CS 及 EIP 寄存器被压入栈内。控制权被转移到由中断向量指定的中断处理程序。在中断处理程序结束时，IRET 指令又把控制权送回到刚才执行被中断的地方。

3．调用溢出处理程序的指令 INTO

指令格式：INTO

说明：在溢出标志为 1 时，INTO 调用中断向量为 4 的异常处理程序。EFLAG、CS 及 EIP 寄存器被压入栈内。控制权被转移到由中断向量 4 指定的异常处理程序。在中断处理程序结束时，IRET 指令又把控制权送回到刚才执行被中断的地方。

4．中断返回指令 IRET

指令格式：IRET

说明：IRET 与中断调用过程相反：它将 EIP、CS 及 EFLAGS 寄存器内容从栈中弹出，并将控制权返回到发生中断的地方。IRET 用在中断处理程序的结束处。

5．加载中断描述符表的指令 LIDT

格式：LIDT　48 位的伪描述符

说明：LIDT 将指令中给定的 48 位伪描述符装入中断描述符寄存器 IDTR。伪描述符和中断描述符表寄存器的结构相同，都是由两部分组成：在低字（低 16 位）中装的是界限，在高双字（高 32 位）中装的是基址。这条指令只能出现在操作系统的代码中。

中断或异常处理程序执行的最后一条指令是返回指令 IRET。这条指令将使 CPU 进行如下操作后，把控制权转交给被中断的进程。

- 从中断处理程序的内核栈中恢复相应寄存器的值。如果一个硬件错码被压入堆栈，则先弹出这个值，然后，依次将 EIP、CS 及 EFLSG 从栈中弹出。
- 检查中断或异常处理程序的 CPL 是否等于 CS 中的最低两位，如果是，这就意味着被中断的进程与中断处理程序都处于内核态，也就是没有更换堆栈，因此，IRET 终止执行，返回到被中断的进程。否则，转入下一步。
- 从栈中装载 SS 和 ESP 寄存器，返回到用户态堆栈。
- 检查 DS、ES、FS 和 GS 四个段寄存器的内容，看它们包含的选择符是否是一个段选择符，并且其 DPL 是否小于 CPL。如果是，就清除其内容。这么做的原因是为了禁止用户态的程序（CPL=3）利用内核曾用过的段寄存器（DPL=0）。如果不这么做，怀有恶意的用户就可能利用这些寄存器来访问内核的地址空间。

3.2　中断描述符表的初始化

通过上面的介绍，我们知道了 Intel 微处理器对中断和异常所做的工作。下面，我们从操作系统的角度来对中断描述符表的初始化给予描述。

Linux 内核在系统的初始化阶段要进行大量的初始化工作，其与中断相关的工作有：初始化可编程控制器 8259A；将中断向量 IDT 表的起始地址装入 IDTR 寄存器，并初始化表中的每一项。这些操作的完成将在本节进行具体描述。

用户进程可以通过 INT 指令发出一个中断请求，其中断请求向量在 0～255 之间。为了防止用户使用 INT 指令模拟非法的中断和异常，必须对 IDT 表进行谨慎的初始化。其措施之一就是将中断门或陷阱门中的 DPL 域置为 0。如果用户进程确实发出了这样一个中断请求，CPU 会检查出其 CPL（3）与 DPL（0）有冲突，因此产生一个“通用保护”异常。

但是，有时候必须让用户进程能够使用内核所提供的功能（比如系统调用），也就是说从用户空间进入内核空间，这可以通过把中断门或陷阱门的 DPL 域置为 3 来达到。

3.2.1 外部中断向量的设置

前面我们已经提到，Linux 把向量 0～31 分配给异常和非屏蔽中断，而把 32～47 之间的向量分配给可屏蔽中断，可屏蔽中断的向量是通过对中断控制器的编程来设置的。前面介绍了 8259A 中断控制器，下面我们通过对其初始化的介绍，来了解如何设置中断向量。

8259A 通过两个端口来进行数据传送，对于单块的 8259A 或者是级连中的 8259A_1 来说，这两个端口是 0x20 和 0x21。对于 8259A_2 来说，这两个端口是 0xA0 和 0xA1。8259A 有两种编程方式，一是初始化方式，二是工作方式。在操作系统启动时，需要对 8959A 做一些初始化工作，这就是初始化方式编程。

先简单介绍一下 8259A 内部的 4 个中断命令字（ICW）寄存器的功能，它们都是用来启动初始化编程的。

- ICW1：初始化命令字。
- ICW2：中断向量寄存器，初始化时写入高 5 位作为中断向量的高五位，然后在中断响应时由 8259 根据中断源（哪个管脚）自动填入形成完整的 8 位中断向量（或叫中断类型号）。
- ICW3：8259 的级连命令字，用来区分主片和从片。
- ICW4：指定中断嵌套方式、数据缓冲选择、中断结束方式和 CPU 类型。

8259A 初始化的目的是写入有关命令字，8259A 内部有相应的寄存器来锁存这些命令字，以控制 8259A 工作。有关的硬件知识笔者就不详细描述了，请读者查阅有关可编程中断控制器的资料，我们只具体把 Linux 对 8259A 的初始化讲解一下，代码在 /arch/i386/kernel/i8259.c 的函数 init_8259A()中:

```
outb（0xff, 0x21）；        /* 送数据到工作寄存器 OCW1（又称中断屏蔽字），
  屏蔽所有外部中断, 因为此时系统尚未初始化完毕,
  outb（0xff, 0xA1）；       不能接收任何外部中断请求  */


        outb_p（0x11, 0x20）；  / *送 0x11 到 ICW1（通过端口 0x20），启动初始化编程。0x11 表示外部中断请求信号为上升沿有效，系统中有多片 8295A 级连，还表示要向 ICW4 送数据 * /


        outb_p（0x20 + 0, 0x21）; /* 送 0x20 到 ICW2，写入高 5 位作为中断向量的高 5 位，低 3 位根据中断源（管脚）填入中断号 0～7，因此把 IRQ0-7 映射到向量 0x20-0x27 */


        outb_p（0x04, 0x21）; /* 送 0x04 到 ICW3，ICW3 是 8259 的级连命令字， 0x04 表示 8259A-1 是主片 */
        outb_p（0x11, 0xA0）；  /* 用 ICW1 初始化 8259A-2 */
        outb_p（0x20 + 8, 0xA1）; /* 用 ICW2 把 8259A-2 的 IRQ0-7 映射到 0x28-0x2f */
        outb_p（0x02, 0xA1）；   /* 送 0x04 到 ICW3。表示 8259A-2 是从片，并连
```

接在 8259A_1 的 2 号管脚上*/

outb_p（0x01, 0xA1）; /* 把 0x01 送到 ICW4 */

最后一句有 4 方面含义：①中断嵌套方式为一般嵌套方式。当某个中断正在服务时，本级中断及更低级的中断都被屏蔽，只有更高级的中断才能响应。注意，这对于多片 8259A 级连的中断系统来说，当某从片中一个中断正在服务时，主片即将这个从片的所有中断屏蔽，所以此时即使本片有比正在服务的中断级别更高的中断源发出请求，也不能得到响应，即不能中断嵌套。②8259A 数据线和系统总线之间不加三态缓冲器。一般来说，只有级连片数很多时才用到三态缓冲器；③中断结束方式为正常方式（非自动结束方式）。即在中断服务结束时（中断服务程序末尾），要向 8259A 芯片发送结束命令字 EOI（送到工作寄存器 OCW2 中），于是中断服务寄存器 ISR 中的当前服务位被清 0，EOI 命令字的格式有多种，在此不详述；④CPU 类型为 x86 系列。

outb_p()函数就是把第一个操作数拷贝到由第二个操作数指定的 I/O 端口，并通过一个空操作来产生一个暂停。

这里介绍了 8259A 初始化的主要工作。最后要说明的是：IBM PC 机的 BIOS 中固化有对中断控制器的初始化程序段，在计算机加电时，这段程序自动执行，读者感兴趣可以查阅资料看看它的源代码。典型的 PC 机将外部中断的中断向量分配为：08H~0FH，70H~77H。但是 Linux 对 8259A 作了重新初始化，修改了外部中断的中断向量的分配（20H~2FH），使中断向量的分配更加合理。

3.2.2　中断描述符表 IDT 的预初始化

当计算机运行在实模式时，IDT 被初始化并由 BIOS 使用。然而，一旦真正进入了 Linux 内核，IDT 就被移到内存的另一个区域，并进行进入实模式的初步初始化。

1．中断描述表寄存器 IDTR 的初始化

用汇编指令 LIDT 对中断向量表寄存器 IDTR 进行初始化，其代码在 arch/i386/boot/setup.S 中：

```
lidt    idt_48                  # load idt with 0,0
…
    idt_48:
         .word   0                  # idt limit = 0
         .word   0, 0               # idt base = 0L
```

2．把 IDT 表的起始地址装入 IDTR

用汇编指令 LIDT 装入 IDT 的大小和它的地址（在 arch/i386/kernel/head.S 中）：

```
#define IDT_ENTRIES     256
.globl SYMBOL_NAME（idt）

lidt idt_descr
…
idt_descr:
        .word IDT_ENTRIES*8-1           # idt contains 256 entries
```

```
SYMBOL_NAME（idt）:
        .long SYMBOL_NAME（idt_table）
```

其中 idt 为一个全局变量，内核对这个变量的引用就可以获得 IDT 表的地址。表的长度为 256×8＝2048 字节。

3．用 setup_idt()函数填充 idt_table 表中的 256 个表项

我们首先要看一下 idt_table 的定义（在 arch/i386/kernel/traps.c 中）：

```
  struct desc_struct idt_table[256] __attribute__（（__section__（".data.idt"））） ={{0,
0}, };
```

desc_struct 结构定义为：

```
struct desc_struct  {
 unsigned long a,b   }
```

对 idt_table 变量还定义了其属性（__attribute__），__section__是汇编中的“节”，指定了 idt_table 的起始地址存放在数据节的 idt 变量中，如上面第“2.把 IDT 表的起始地址装入 IDTR”所述。

在对 idt_table 表进行填充时，使用了一个空的中断处理程序 ignore_int()。因为现在处于初始化阶段，还没有任何中断处理程序，因此用这个空的中断处理程序填充每个表项。ignore_int()是一段汇编程序（在 head.S 中）：

```
ignore_int:
        cld        #方向标志清 0，表示串指令自动增长它们的索引寄存器（esi 和
edi）
        pushl %eax
        pushl %ecx
        pushl %edx
        pushl %es
        pushl %ds
        movl $（__KERNEL_DS）,%eax
        movl %eax,%ds
        movl %eax,%es
        pushl $int_msg
        call SYMBOL_NAME（printk）
        popl %eax
        popl %ds
        popl %es
        popl %edx
        popl %ecx
        popl %eax
        iret
int_msg:
      .asciz "Unknown interrupt\n"
       ALIGN
```

该中断处理程序模仿一般的中断处理程序，执行如下操作：

- 在栈中保存一些寄存器的值；
- 调用 printk()函数打印“Unknown interrupt”系统信息；

- 从栈中恢复寄存器的内容；
- 执行 iret 指令以恢复被中断的程序。

实际上，ignore_int()处理程序应该从不执行。如果在控制台或日志文件中出现了“Unknown interrupt”消息，说明要么是出现了一个硬件问题（一个 I/O 设备正在产生没有预料到的中断），要么就是出现了一个内核问题（一个中断或异常未被恰当地处理）。

最后，我们来看 setup_idt()函数如何对 IDT 表进行填充：

```
/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It doesn't actually load
 *  idt - that can be done only after paging has been enabled
 *  and the kernel moved to PAGE_OFFSET. Interrupts
 *  are enabled elsewhere, when we can be relatively
 *  sure everything is ok.
 */
setup_idt:
        lea ignore_int,%edx    /*计算 ignore_int 地址的偏移量,并将其装入%edx*/
        movl $ (__KERNEL_CS << 16),%eax  /* selector = 0x0010 = cs */
        movw %dx,%ax
        movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */

        lea SYMBOL_NAME (idt_table),%edi
        mov $256,%ecx
rp_sidt:
        movl %eax, (%edi)
        movl %edx,4 (%edi)
        addl $8,%edi
        dec %ecx
        jne rp_sidt
        ret
```

这段程序的理解要对照门描述符的格式。8 个字节的门描述符放在两个 32 位寄存器 eax 和 edx 中，如图 3.4 所示，从 rp_sidt 开始的那段程序是循环填充 256 个表项。

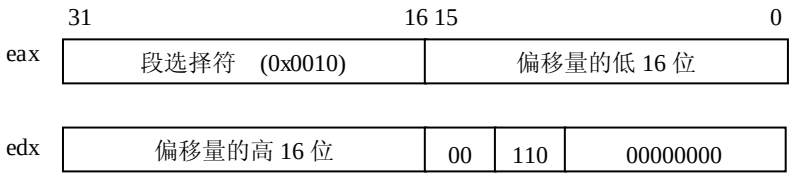


图 3.4　门描述符存放在两个 32 位的寄存器中

3.2.3　中断向量表的最终初始化

在对中断描述符表进行预初始化后，内核将在启用分页功能后对 IDT 进行第二遍初始化，也就是说，用实际的陷阱和中断处理程序替换这个空的处理程序。一旦这个过程完成，对于

每个异常，IDT 都由一个专门的陷阱门或系统门，而对每个外部中断，IDT 都包含专门的中断门。

1．IDT 表项的设置

IDT 表项的设置是通过_set_gaet()函数实现的，这与 IDT 表的预初始化比较相似，但这里使用的是嵌入式汇编，因此，理解起来比较困难。在此，我们给出函数源码（在 traps.c 中）及其解释：

```
#define _set_gate (gate_addr,type,dpl,addr)  \
do { \
   int __d0, __d1; \
   __asm__ __volatile__  ("movw %%dx,%%ax\n\t" \
         "movw %4,%%dx\n\t" \
         "movl %%eax,%0\n\t" \
        "movl %%edx,%1" \
         :"=m"  (* ( (long *)  (gate_addr) ) ), \
          "=m"  (* (1+ (long *)  (gate_addr) ) ), "=&a"  (__d0), "=&d"  (__d1)  \
         :"i"  ( (short)  (0x8000+ (dpl<<13) + (type<<8) ) ), \
          "3"  ( (char *)  (addr) ),"2"  (__KERNEL_CS << 16) ); \
} while (0)
```

这是一个带参数的宏定义，其中，gate_addr 是门的地址，type 为门类型，dpl 为请求特权级，addr 为中断处理程序的地址。对这段嵌入式汇编代码的说明如下：

- 输出部分有 4 个变量，分别与%1、%2、%3 及%4 相结合，其中，%0 与 gate_addr 结合，1%与（gate_aggr＋1）结合，这两个变量都存放在内存；2%与局部变量__d0 结合，存放在 eax 寄存器中；3%与__d1 结合，存放在 edx 寄存器中。
- 输入部分有 3 个变量。由于输出部分已定义了 0%～3%，因此，输入部分的第一个变量为 4%，其值为“0x8000+（dpl<<13）+（type<<8”，而后面两个变量分别等价于输出部分的%3（edx）和 2%（eax），其值分别为“addr”和“__KERNEL_CS << 16”
- 有了参数的这种对照关系，再参考前面的 set_idt()函数，就不难理解那 4 条 mov 语句了。

下面我们来看如何调用_set_get()函数来给 IDT 插入门:

```
void set_intr_gate (unsigned int n, void *addr)
{
        _set_gate (idt_table+n,14,0,addr) ;
}
```

在第 *n* 个表项中插入一个中断门。这个门的段选择符设置成代码段的选择符(__KERNEL_CS),DPL 域设置成 0,14 表示 D 标志位为 1 而类型码为 110,所以 set_intr_gate()设置的是中断门，偏移域设置成中断处理程序的地址 addr。

```
static void __init set_trap_gate (unsigned int n, void *addr)
{
        _set_gate (idt_table+n,15,0,addr) ;
}
```

在第 *n* 个表项中插入一个陷阱门。这个门的段选择符设置成代码段的选择符，DPL 域设置成 0，15 表示 D 标志位为 1 而类型码为 111，所以 set_trap_gate()设置的是陷阱门，偏移域设置成异常处理程序的地址 addr。

```
static void __init set_system_gate (unsigned int n, void *addr)
```

```
{
        _set_gate (idt_table+n,15,3,addr) ;
}
```

在第 *n* 个表项中插入一个系统门。这个门的段选择符设置成代码段的选择符，DPL 域设置成 3，15 表示 D 标志位为 1 而类型码为 111，所以 set_system_gate()设置的也是陷阱门，但因为 DPL 为 3，因此，系统调用在用户空间可以通过“INT0X80”顺利穿过系统门，从而进入内核空间。

2．对陷阱门和系统门的初始化

trap_init()函数就是设置中断描述符表开头的 19 个陷阱门，如前所说，这些中断向量都是 CPU 保留用于异常处理的：

```
        set_trap_gate (0,&divide_error) ;
        set_trap_gate (1,&debug) ;
        set_intr_gate (2,&nmi) ;
        set_system_gate (3,&int3) ;       /* int3-5 can be called from all */
        set_system_gate (4,&overflow) ;
        set_system_gate (5,&bounds) ;
        set_trap_gate (6,&invalid_op) ;
        set_trap_gate (7,&device_not_available) ;
        set_trap_gate (8,&double_fault) ;
        set_trap_gate (9,&coprocessor_segment_overrun) ;
        set_trap_gate (10,&invalid_TSS) ;
        set_trap_gate (11,&segment_not_present) ;
        set_trap_gate (12,&stack_segment) ;
        set_trap_gate (13,&general_protection) ;
        set_intr_gate (14,&page_fault) ;
        set_trap_gate (15,&spurious_interrupt_bug) ;
        set_trap_gate (16,&coprocessor_error) ;
        set_trap_gate (17,&alignment_check) ;
        set_trap_gate (18,&machine_check) ;
        set_trap_gate (19,&simd_coprocessor_error) ;

        set_system_gate (SYSCALL_VECTOR,&system_call) ;
```

在对陷阱门及系统门设置以后，我们来看一下中断门的设置。

3．中断门的设置

下面介绍的相关代码均在 arch/I386/kernel/i8259.c 文件中，其中中断门的设置是由 init_IRQ（ ）函数中的一段代码完成的：

```
for  (i = 0; i< NR_IRQS; i++)  {
                int vector = FIRST_EXTERNAL_VECTOR + i;
               if  (vector != SYSCALL_VECTOR)
                       set_intr_gate (vector, interrupt[i]) ;
```

其含义比较明显：从 FIRST_EXTERNAL_VECTOR 开始，设置 NR_IRQS 个 IDT 表项。常数 FIRST_EXTERNAL_VECTOR 定义为 0x20，而 NR_IRQS 则为 224，即中断门的个数。注意，必须跳过用于系统调用的向量 0x80，因为这在前面已经设置好了。

这里，中断处理程序的入口地址是一个数组 interrupt[]，数组中的每个元素是指向中

断处理函数的指针。我们来看一下编码的作者如何巧妙地避开了繁琐的文字录入，而采用统一的方式处理多个函数名。

```
#define IRQ(x,y)  \
         IRQ##x##y##_interrupt

#define IRQLIST_16(x)  \
        IRQ(x,0), IRQ(x,1), IRQ(x,2), IRQ(x,3), \
        IRQ(x,4), IRQ(x,5), IRQ(x,6), IRQ(x,7), \
        IRQ(x,8), IRQ(x,9), IRQ(x,a), IRQ(x,b), \
        IRQ(x,c), IRQ(x,d), IRQ(x,e), IRQ(x,f)
void (*interrupt[NR_IRQS])(void) = IRQLIST_16(0x0)
```

其中，“##”的作用是把字符串连接在一起。经过 gcc 预处理，IRQLIST_16（0x0）被替换为 IRQ0x00_interrupt，IRQ0x01_interrupt，IRQ0x02_interrupt…IRQ0x0f_interrupt。

到此为止，我们已经介绍了 15 个陷阱门、4 个系统门和 16 个中断门的设置。内核代码中还有对其他中断门的设置，在此就不一一介绍。

3.3 异常处理

Linux 利用异常来达到两个截然不同的目的：

- 给进程发送一个信号以通报一个反常情况；
- 处理请求分页。

对于第一种情况，例如，如果进程执行了一个被 0 除的操作，CPU 则会产生一个“除法错误”异常，并由相应的异常处理程序向当前进程发送一个 SIGFPE 信号。当前进程接收到这个信号后，就要采取若干必要的步骤，或者从错误中恢复，或者终止执行（如果这个信号没有相应的信号处理程序）。

内核对异常处理程序的调用有一个标准的结构，它由以下 3 部分组成：

- 在内核栈中保存大多数寄存器的内容（由汇编语言实现）；
- 调用 C 编写的异常处理函数；
- 通过 ret_from_exception()函数从异常退出。

3.3.1 在内核栈中保存寄存器的值

所有异常处理程序被调用的方式比较相似，因此，我们用 handler_name 来表示一个通用的异常处理程序的名字（实际名字都出现在表 3.1 中）。进入异常处理程序的汇编指令在 arch/I386/kernel/entry.S 中：

```
handler_name:
    pushl $0  /* only for some exceptions */
    pushl $do_handler_name
jmp error_code

例如： overflow:
              pushl $0
```

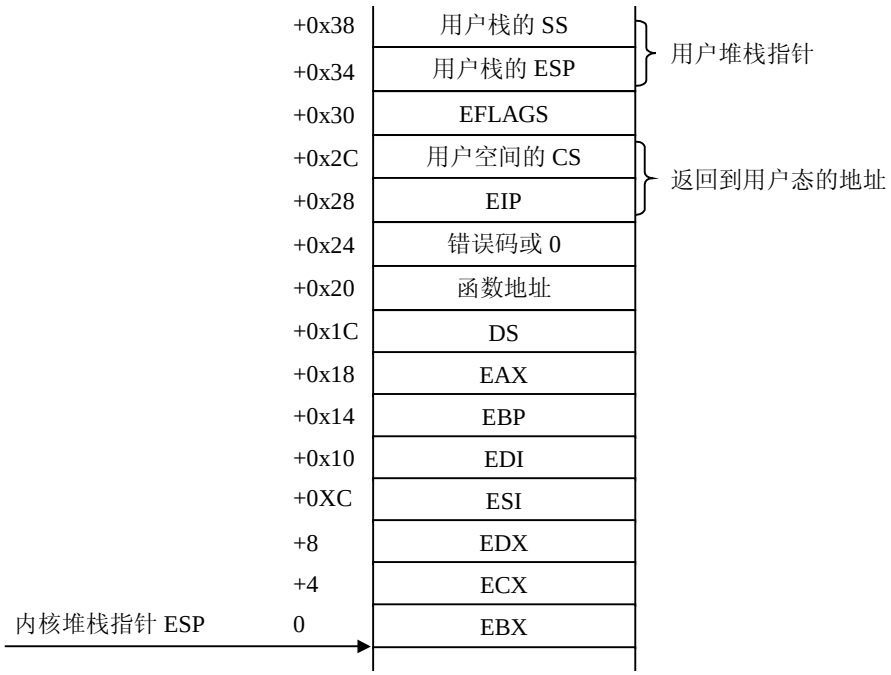
```
        pushl $ do_overflow
        jmp error_code
```

当异常发生时，如果控制单元没有自动地把一个硬件错误代码插入到栈中，相应的汇编语言片段会包含一条 pushl $0 指令，在栈中垫上一个空值；如果错误码已经被压入堆栈，则没有这条指令。然后，把异常处理函数的地址压进栈中，函数的名字由异常处理程序名与 do_ 前缀组成。

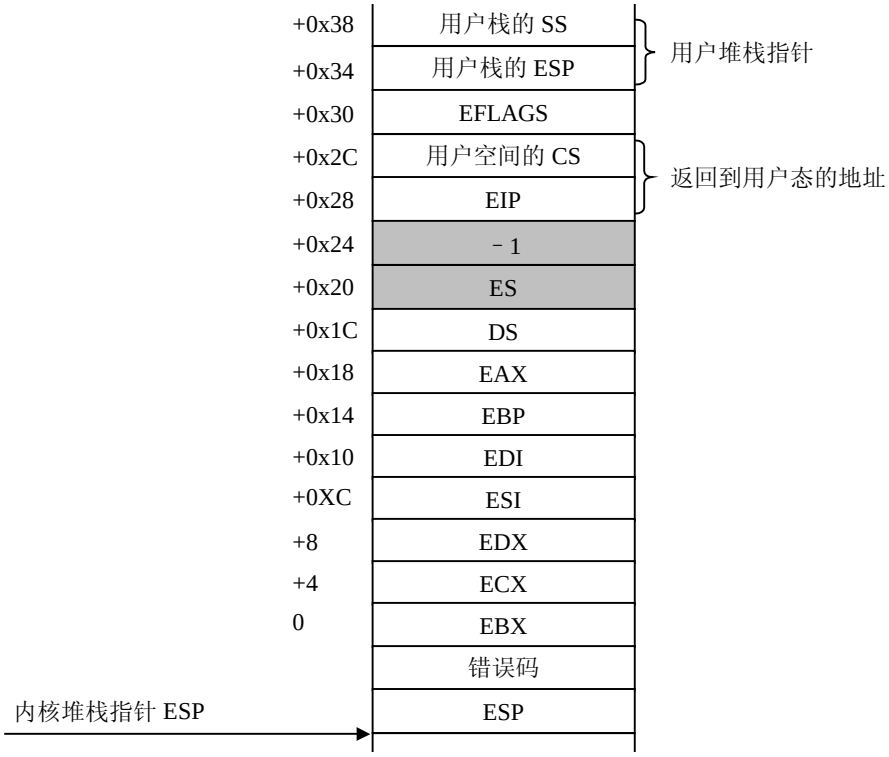
标号为 error_code 的汇编语言片段对所有的异常处理程序都是相同的，除了“设备不可用”这一个异常。这段代码实际上是为异常处理程序的调用和返回进行相关的操作，代码如下：

```
error_code:
      pushl %ds
      pushl %eax
      xorl %eax,%eax
      pushl %ebp
      pushl %edi              #把 C 函数可能用到的寄存器都保存在栈中
      pushl %esi
      pushl %edx
      decl %eax            #eax = -1
      pushl %ecx
      pushl %ebx
      cld               # 清 eflags 的方向标志，以确保 edi 和 esi 寄存器的值自动增加
      movl %es,%ecx
      movl ORIG_EAX (%esp), %esi       # get the error code, ORIG_EAX= 0x24
      movl ES (%esp), %edi              # get the function address, ES = 0x20
      movl %eax, ORIG_EAX (%esp)       #把栈中的这个位置置为-1
      movl %ecx, ES (%esp)
      movl %esp,%edx
      pushl %esi                      # push the error code
      pushl %edx                     # push the pt_regs pointer
      movl $ (__KERNEL_DS),%edx
      movl %edx,%ds            #把内核数据段选择符装入 ds 寄存器
      movl %edx,%es
      GET_CURRENT (%ebx)       #ebx 中存放当前进程 task_struct 结构的地址
      call *%edi                      #调用这个异常处理程序
      addl $8,%esp
     jmp ret_from_exception
```

图 3.5 给出了从用户进程进入异常处理程序时内核堆栈的变化示意图。



(a) 进入异常处理程序时内核堆栈示意图



(b) 异常处理程序被调用后堆栈的示意图

图 3.5　进入异常后内核堆栈的变化

3.3.2　中断请求队列的初始化

由于硬件的限制，很多外部设备不得不共享中断线，例如，一些 PC 配置可以把同一条中断线分配给网卡和图形卡。由此看来，让每个中断源都必须占用一条中断线是不现实的。所以，仅用中断描述符表并不能提供中断产生的所有信息，内核必须对中断线给出进一步的描述。在 Linux 设计中，专门为每个中断请求 IRQ 设置了一个队列，这就是我们所说的中断请求队列。

注意，中断线、中断请求（IRQ）号及中断向量之间的关系为：中断线是中断请求的一种物理描述，逻辑上对应一个中断请求号（或简称中断号），第 n 个中断号（IRQn）的缺省中断向量是 n+32。

3.3.3　中断请求队列的数据结构

如前所述，在 256 个中断向量中，除了 32 个分配给异常外，还有 224 个作为中断向量。对于每个 IRQ，Linux 都用一个 irq_desc_t 数据结构来描述，我们把它叫做 IRQ 描述符，224 个 IRQ 形成一个数组 irq_desc[]，其定义在/include/linux/irq.h 中：

```
 /*
  * This is the "IRQ descriptor", which contains various information
  * about the irq, including what kind of hardware handling it has,
  * whether it is disabled etc etc.
  *
  * Pad this out to 32 bytes for cache and indexing reasons.
  */
typedef struct {
        unsigned int status;            /* IRQ status */
        hw_irq_controller *handler;
        struct irqaction *action;       /* IRQ action list */
        unsigned int depth;            /* nested irq disables */
        spinlock_t lock;
} ____cacheline_aligned irq_desc_t;

extern irq_desc_t irq_desc [NR_IRQS];
```

编码作者对这个数据结构给出了一定的解释，“____cacheline_aligned”表示这个数据结构的存放按 32 字节（高速缓存行的大小）进行对齐，以便于将来存放在高速缓存并容易存取。下面对这个数据结构的各个域给予描述。

status

描述 IRQ 中断线状态的一组标志（在 irq.h 中定义）,其具体含义及应用将在 do_IRQ()函数中介绍。

handler

指向 hw_interrupt_type 描述符，这个描述符是对中断控制器的描述，下面会给出具体解释。

action

指向一个单向链表的指针，这个链表就是对中断服务例程进行描述的 irqaction 结构，后面将给予具体描述。

depth

如果启用这条 IRQ 中断线，depth 则为 0，如果禁用这条 IRQ 中断线不止一次，则为一个正数。每当调用一次 disable_irq（ ），该函数就对这个域的值加 1；如果 depth 等于 0，该函数就禁用这条 IRQ 中断线。相反，每当调用 enable_irq（ ）函数时，该函数就对这个域的值减 1；如果 depth 变为 0，该函数就启用这条 IRQ 中断线。

1．IRQ 描述符的初始化

在系统初始化期间，init_ISA_irqs()函数对 IRQ 数据结构（或叫描述符）的域进行初始化（参见 i8258.c）：

```
for (i = 0; i < NR_IRQS; i++) {
            irq_desc[i].status = IRQ_DISABLED;
            irq_desc[i].action = 0;
            irq_desc[i].depth = 1;

            if (i < 16) {
                    /*
                     * 16 old-style INTA-cycle interrupts:
                     */
                    irq_desc[i].handler = &i8259A_irq_type;
            } else {
                    /*
                     * 'high' PCI IRQs filled in on demand
                     */
                    irq_desc[i].handler = &no_irq_type;
            }
    }
```

从这段程序可以看出，初始化时，让所有的中断线都处于禁用状态；每条中断线上还没有任何中断服务例程（action 为 0）；因为中断线被禁用，因此 depth 为 1；对中断控制器的描述分为两种情况，一种就是通常所说的 8259A，另一种是其他控制器。

然后，更新中断描述符表 IDT，如 3.2.3 节所述，用最终的中断门来代替临时使用的中断门。

2．中断控制器描述符 hw_interrupt_type

这个描述符包含一组指针，指向与特定中断控制器电路（PIC）打交道的低级 I/O 例程，定义如下：

```
/*
* Interrupt controller descriptor. This is all we need
* to describe about the low-level hardware.
*/
struct hw_interrupt_type {
        const char * typename;
        unsigned int (*startup) (unsigned int irq);
        void (*shutdown) (unsigned int irq);
       void (*enable) (unsigned int irq);
       void (*disable) (unsigned int irq);
       void (*ack) (unsigned int irq);
       void (*end) (unsigned int irq);
```

```
        void  (*set_affinity)  (unsigned int irq, unsigned long mask) ;
};

typedef struct hw_interrupt_type  hw_irq_controller;
```

Linux 除了支持本章前面已提到的 8259A 芯片外，也支持其他的 PIC 电路，如 SMP IO-APIC、PIIX4 的内部 8259 PIC 及 SGI 的 Visual Workstation Cobalt （IO-）APIC。但是，为了简单起见，我们在本章假定，我们的计算机是有两片 8259A PIC 的单处理机，它提供 16 个标准的 IRQ。在这种情况下，有 16 个 irq_desc_t 描述符，其中每个描述符的 handler 域指向 8259A_irq _type 变量。对其进行如下的初始化：

```
struct hw_interrupt_type i8259A_irq_type = {
          "XT-PIC",
          startup_8259A_irq,
          shutdown_8259A_irq,
          do_8259A_IRQ,
          enable_8259A_irq,
          disable_8259A_irq
    };
```

在这个结构中的第一个域“XT-PIC”是一个名字。接下来，8259A_irq_type 包含的指针指向 5 个不同的函数，这些函数就是对 PIC 编程的函数。前两个函数分别启动和关闭这个芯片的中断线。但是，在使用 8259A 芯片的情况下，这两个函数的作用与后两个函数是一样的，后两个函数是启用和禁用中断线。后面在对 do_IRQ 描述时具体描述 do_8259A_IRQ（　）函数。

3．中断服务例程描述符 irqaction

在 IRQ 描述符中我们看到指针 action 的结构为 irqaction，它是为多个设备能共享一条中断线而设置的一个数据结构。在 include/linux/interrupt.h 中定义如下:

```
  struct irqaction {
        void  (*handler)  (int, void *, struct pt_regs *) ;
         unsigned long flags;
        unsigned long mask;
        const char *name;
        void *dev_id;
        struct irqaction *next;
  };
```

这个描述符包含下列域。

handler

指向一个具体 I/O 设备的中断服务例程。这是允许多个设备共享同一中断线的关键域。

flags

用一组标志描述中断线与 I/O 设备之间的关系。

SA_INTERRUPT

中断处理程序必须以禁用中断来执行。

SA_SHIRQ

该设备允许其中断线与其他设备共享。

SA_SAMPLE_RANDOM

可以把这个设备看作是随机事件发生源；因此，内核可以用它做随机数产生器（用户可

以从/dev/random 和/dev/urandom 设备文件中取得随机数而访问这种特征)。

SA_PROBE

内核在执行硬件设备探测时正在使用这条中断线。

name

I/O 设备名（读取/proc/interrupts 文件，可以看到，在列出中断号时也显示设备名)。

dev_id

指定 I/O 设备的主设备号和次设备号。

next

指向 irqaction 描述符链表的下一个元素。共享同一中断线的每个硬件设备都有其对应的中断服务例程，链表中的每个元素就是对相应设备及中断服务例程的描述。

4．中断服务例程

我们这里提到的中断服务例程（Interrupt Service Routine）与以前所提到的中断处理程序（Interrupt handler）是不同的概念。具体来说，中断处理程序相当于某个中断向量的总处理程序，例如 IRQ0x05_interrupt()，是中断号 5（向量为 37）的总处理程序，如果这个 5 号中断由网卡和图形卡共享，则网卡和图形卡分别有其相应的中断服务例程。每个中断服务例程都有相同的参数：

IRQ：中断号；

dev_id: 设备标识符，其类型为 void*；

regs: 指向内核堆栈区的指针，堆栈中存放的是中断发生后所保存的寄存器，有关 pt_regs 结构的具体内容将在后面介绍。

在实际中，大多数中断服务例程并不使用这些参数。

3.3.2 中断请求队列的初始化

在 IDT 表初始化完成之初，每个中断服务队列还为空。此时，即使打开中断且某个外设中断真的发生了，也得不到实际的服务。因为 CPU 虽然通过中断门进入了某个中断向量的总处理程序，例如 IRQ0x05_interrupt()，但是，具体的中断服务例程（如图形卡的）还没有挂入中断请求队列。因此，在设备驱动程序的初始化阶段，必须通过 request_irq()函数将对应的中断服务例程挂入中断请求队列。

request_irq()函数的代码在 / arch/i386/kernel/irq.c 中：

```
/*
*      request_irq - allocate an interrupt line
*      @irq: Interrupt line to allocate
*      @handler: Function to be called when the IRQ occurs
*      @irqflags: Interrupt type flags
*      @devname: An ascii name for the claiming device
*      @dev_id: A cookie passed back to the handler function
*
*      This call allocates interrupt resources and enables the
*      interrupt line and IRQ handling. From the point this
*      call is made your handler function may be invoked. Since
*      your handler function must clear any interrupt the board
```

```
 *      raises, you must take care both to initialise your hardware
 *      and to set up the interrupt handler in the right order.
 *
 *      Dev_id must be globally unique. Normally the address of the
 *      device data structure is used as the cookie. Since the handler
 *      receives this value it makes sense to use it.
 *
 *      If your interrupt is shared you must pass a non NULL dev_id
 *      as this is required when freeing the interrupt.
 *
 *      Flags:
 *
 *      SA_SHIRQ                Interrupt is shared
 *
 *      SA_INTERRUPT            Disable local interrupts while processing
 *
 *      SA_SAMPLE_RANDOM        The interrupt can be used for entropy
 *
 */

int request_irq (unsigned int irq,
                 void  (*handler)  (int, void *, struct pt_regs *) ,
                 unsigned long irqflags,
                 const char * devname,
                 void *dev_id)
{
        int retval;
        struct irqaction * action;

#if 1
        /*
        * Sanity-check: shared interrupts should REALLY pass in
        * a real dev-ID, otherwise we'll have trouble later trying
        * to figure out which interrupt is which  (messes up the
        * interrupt freeing logic etc) .
        */
        if  (irqflags & SA_SHIRQ)  {
               if  (!dev_id)
                       printk("Bad boy: %s (at 0x%x) called us without a dev_id!\n", devname,
(&irq) [-1]) ;
        }
#endif

        if  (irq >= NR_IRQS)
                return -EINVAL;
        if  (!handler)
               return -EINVAL;

       action =  (struct irqaction *)
                        kmalloc (sizeof (struct irqaction) , GFP_KERNEL) ;
        if  (!action)
               return -ENOMEM;
```

```
        action->handler = handler;
        action->flags = irqflags;
        action->mask = 0;
        action->name = devname;       /*对 action 进行初始化*/
        action->next = NULL;
        action->dev_id = dev_id;

        retval = setup_irq (irq, action) ;
        if  (retval)
                kfree (action) ;
        return retval;
}
```

编码作者对此函数给出了比较详细的描述。其中主要语句就是对 setup_irq()函数的调用，该函数才是真正对中断请求队列进行初始化的函数（有所简化）：

```
int setup_irq (unsigned int irq, struct irqaction * new)
{
        int shared = 0;
        unsigned long flags;
        struct irqaction *old, **p;
        irq_desc_t *desc = irq_desc + irq;       /*获得 irq 的描述符*/

          /* 对中断请求队列的操作必须在临界区中进行 */
        spin_lock_irqsave (&desc->lock,flags) ;    /*进入临界区*/
        p = &desc->action;    /*让 p 指向 irq 描述符的 action 域，即 irqaction 链表的首部*/
       if  ( (old = *p)  != NULL) {   /*如果这个链表不为空*/
                /* Can't share interrupts unless both agree to */
               if  (! (old->flags & new->flags & SA_SHIRQ) )  {
                       spin_unlock_irqrestore (&desc->lock,flags) ;
                     return -EBUSY;
               }

               /* 把新的中断服务例程加入到 irq 中断请求队列*/
               do {
                       p = &old->next;
                       old = *p;
               } while  (old) ;
               shared = 1;
        }

        *p = new;

        if  (!shared)  {        /*如果 irq 不被共享 */
               desc->depth = 0;     /*启用这条 irq 线*/
               desc->status &= ~ (IRQ_DISABLED | IRQ_AUTODETECT | IRQ_WAITING) ;
               desc->handler->startup (irq) ;  /*即调用 startup_8259A_irq()函数*/
        }
        spin_unlock_irqrestore (&desc->lock,flags) ;    /*退出临界区*/

        register_irq_proc (irq) ;      /*在 proc 文件系统中显示 irq 的信息*/
        return 0;
```

```
}
```

下面我们举例说明对这两个函数的使用。

1．对 register_irq()函数的使用

在驱动程序初始化或者在设备第一次打开时，首先要调用该函数，以申请使用该 irq。其中参数 handler 指的是要挂入到中断请求队列中的中断服务例程。假定一个程序要对 / dev/fd0/（第一个软盘对应的设备）设备进行访问，有两种方式，一是直接访问 / dev/fd0/，另一种是在系统上安装一个文件系统，我们这里假定采用第一种。通常将 IRQ6 分配给软盘控制器，给定这个中断号 6，软盘驱动程序就可以发出下列请求，以将其中断服务例程挂入中断请求队列：

```
request_irq（6, floppy_interrupt,
        SA_INTERRUPT|SA_SAMPLE_RANDOM, "floppy", NULL）;
```

我们可以看到，floppy_interrupt()中断服务例程运行时必须禁用中断（设置了 SA_INTERRUPT 标志），并且不允许共享这个 IRQ（清 SA_SHIRQ 标志）。

在关闭设备时，必须通过调用 free_irq()函数释放所申请的中断请求号。例如，当软盘操作终止时（或者终止对 / dev/fd0/的 I/O 操作，或者卸载这个文件系统），驱动程序就放弃这个中断号：

```
free_irq（6, NULL）;
```

2．对 setup_ irq()函数的使用

在系统初始化阶段，内核为了初始化时钟中断设备 irq0 描述符，在 time_init（　）函数中使用了下面的语句：

```
struct irqaction irq0  =
    {timer_interrupt, SA_INTERRUPT, 0, "timer", NULL,};
    setup_irq（0, &irq0）;
```

首先，初始化类型为 irqaction 的 irq0 变量，把 handler 域设置成 timer_interrupt（　）函数的地址，flags 域设置成 SA_INTERRUPT，name 域设置成"timer"，最后一个域设置成 NULL 以表示没有用 dev_id 值。接下来，内核调用 setup_x86_irq（　），把 irq0 插入到 IRQ0 的中断请求队列：

类似地，内核初始化与 IRQ2 和 IRQ13 相关的 irqaction 描述符，并把它们插入到相应的请求队列中，在 init_IRQ（　）函数中有下面的语句：

```
struct irqaction irq2 =
         {no_action, 0, 0, "cascade", NULL,};
struct irqaction irq13 =
    {    math_error_irq, 0, 0, "fpu", NULL,};
setup_x86_irq（2, &irq2）;
setup_x86_irq（13, &irq13）;
```

3.4　中断处理

通过上面的介绍，我们知道了中断描述符表已被初始化，并具有了相应的内容；对于外

部中断，中断请求队列也已建立。但是，中断处理程序并没有执行，如何执行中断处理程序正是我们本节要关心的主要内容

3.4.1 中断和异常处理的硬件处理。

首先，我们从硬件的角度来看 CPU 如何处理中断和异常。这里假定内核已被初始化，CPU 已从实模式转到保护模式。

当 CPU 执行了当前指令之后，CS 和 EIP 这对寄存器中所包含的内容就是下一条将要执行指令的逻辑地址。在对下一条指令执行前，CPU 先要判断在执行当前指令的过程中是否发生了中断或异常。如果发生了一个中断或异常，那么 CPU 将做以下事情。

- 确定所发生中断或异常的向量 i（在 0～255 之间）。
- 通过 IDTR 寄存器找到 IDT 表，读取 IDT 表第 i 项（或叫第 i 个门）。
- 分两步进行有效性检查：首先是“段”级检查，将 CPU 的当前特权级 CPL（存放在 CS 寄存器的最低两位）与 IDT 中第 i 项段选择符中的 DPL 相比较，如果 DPL（3）大于 CPL（0），就产生一个“通用保护”异常（中断向量 13），因为中断处理程序的特权级不能低于引起中断的程序的特权级。这种情况发生的可能性不大，因为中断处理程序一般运行在内核态，其特权级为 0。然后是“门”级检查，把 CPL 与 IDT 中第 i 个门的 DPL 相比较，如果 CPL 大于 DPL，也就是当前特权级（3）小于这个门的特权级（0），CPU 就不能“穿过”这个门，于是产生一个“通用保护”异常，这是为了避免用户应用程序访问特殊的陷阱门或中断门。但是请注意，这种“门”级检查是针对一般的用户程序，而不包括外部 I/O 产生的中断或因 CPU 内部异常而产生的异常，也就是说，如果产生了中断或异常，就免去了“门”级检查。
- 检查是否发生了特权级的变化。当中断发生在用户态（特权级为 3），而中断处理程序运行在内核态（特权级为 0），特权级发生了变化，所以会引起堆栈的更换。也就是说，从用户堆栈切换到内核堆栈。而当中断发生在内核态时，即 CPU 在内核中运行时，则不会更换堆栈，如图 3.6 所示。

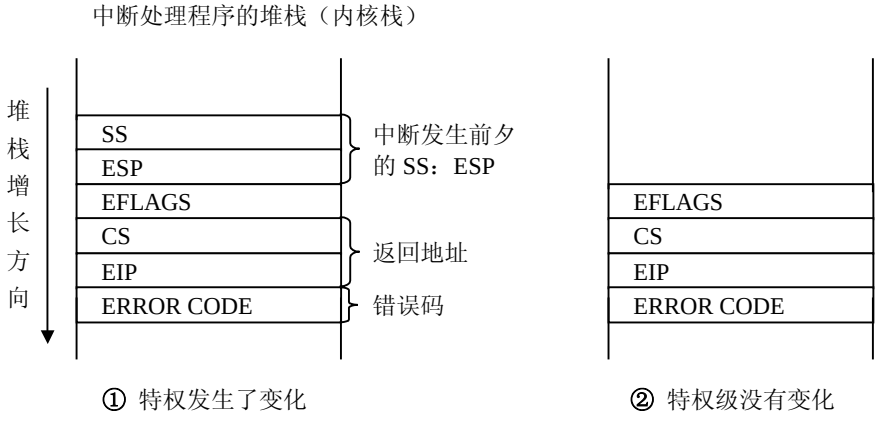


图 3.6 中断处理程序堆栈示意图

从图 3.5 中可以看出，当从用户态堆栈切换到内核态堆栈时，先把用户态堆栈的值压入中

断程序的内核态堆栈中，同时把 EFLAGS 寄存器自动压栈，然后把被中断进程的返回地址压入堆栈。如果异常产生了一个硬件错误码，则将它也保存在堆栈中。如果特权级没有发生变化，则压入栈中的内容如图 3.6 中②。你可能要问，现在 SS:ESP 和 CS:EIP 这两对寄存器的值分别是什么？SS:ESP 的值从当前进程的 TSS 中获得，也就是获得当前进程的内核栈指针，因为此时中断处理程序成为当前进程的一部分，代表当前进程在运行。CS:EIP 的值就是 IDT 表中第 i 项门描述符的段选择符和偏移量的值，此时，CPU 就跳转到了中断或异常处理程序。

3.4.2　Linux 对中断的处理

上面给出了硬件对异常和中断进行处理的一般步骤，下面将概要描述 Linux 对中断的处理，具体的实现过程将在后面介绍。

当一个中断发生时，并不是所有的操作都具有相同的急迫性。事实上，把所有的操作都放进中断处理程序本身并不合适。需要时间长的、非重要的操作应该推后，因为当一个中断处理程序正在运行时，相应的 IRQ 中断线上再发出的信号就会被忽略。更重要的是，中断处理程序是代表进程执行的，它所代表的进程必须总处于 TASK_RUNNING 状态，否则，就可能出现系统僵死情形。因此，中断处理程序不能执行任何阻塞过程，如 I/O 设备操作。因此，Linux 把一个中断要执行的操作分为下面的 3 类。

1．紧急的（Critical）

这样的操作诸如：中断到来时中断控制器做出应答，对中断控制器或设备控制器重新编程，或者对设备和处理器同时访问的数据结构进行修改。这些操作都是紧急的，应该被很快地执行，也就是说，紧急操作应该在一个中断处理程序内立即执行，而且是在禁用中断的状态下。

2．非紧急的（Noncritical）

这样的操作如修改那些只有处理器才会访问的数据结构（例如，按下一个键后，读扫描码）。这些操作也要很快地完成，因此，它们由中断处理程序立即执行，但在启用中断的状态下。

3．非紧急可延迟的（Noncritical deferrable）

这样的操作如，把一个缓冲区的内容拷贝到一些进程的地址空间（例如，把键盘行缓冲区的内容发送到终端处理程序的进程）。这些操作可能被延迟较长的时间间隔而不影响内核操作：有兴趣的进程会等待需要的数据。非紧急可延迟的操作由一些被称为“下半部分”（bottom halves）的函数来执行。我们将在后面讨论“下半部分”。

所有的中断处理程序都执行 4 个基本的操作：

- 在内核栈中保存 IRQ 的值和寄存器的内容；
- 给与 IRQ 中断线相连的中断控制器发送一个应答，这将允许在这条中断线上进一步发出中断请求；
- 执行共享这个 IRQ 的所有设备的中断服务例程（ISR）；
- 跳到 ret_from_intr（　）的地址后终止。

3.4.3 与堆栈有关的常量、数据结构及宏

在中断、异常及系统调用的处理中，涉及一些相关的常量、数据结构及宏，在此先给予介绍（大部分代码在 arch/i386/kernel/entry.S 中）。

1．常量定义

下面这些常量定义了进入中断处理程序时，相关寄存器与堆栈指针（ESP）的相对位置，图 3.7 给出了在相应位置上所保存的寄存器内容。

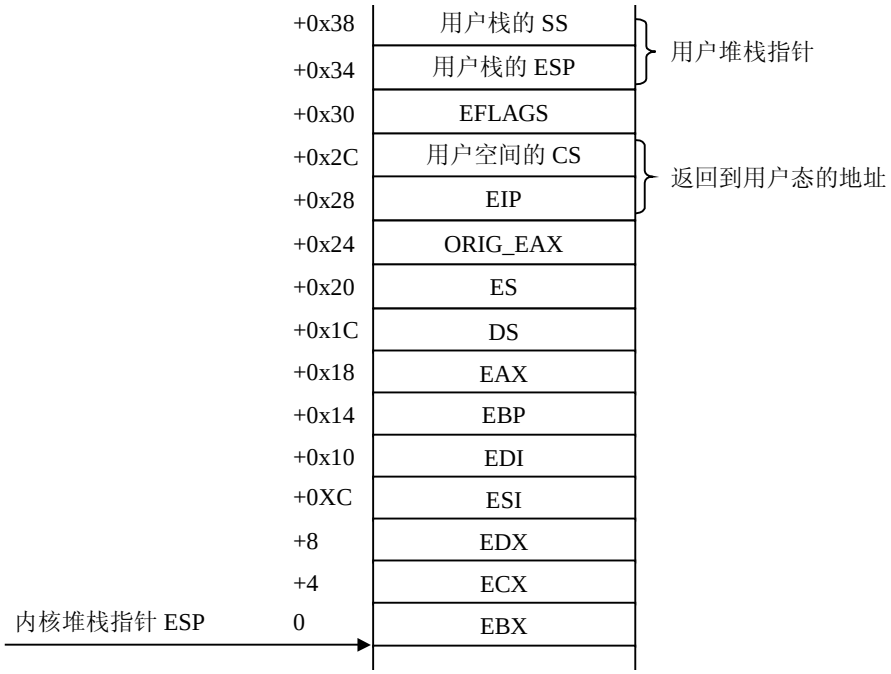


图 3.7 进入中断理程序时内核堆栈示意图

```
EBX = 0x00
ECX= 0x04
EDX= 0x08
ESI= 0x0C
EDI= 0x10
EB = 0x14
EAX= 0x18
DS= 0x1C
ES = 0x20
ORIG_EAX = 0x24
EIP = 0x28
CS  = 0x2C
EFLAGS = 0x30
OLDESP= 0x34
OLDSS = 0x38
```

其中，ORIG_EAX 是 Original eax 之意，其具体含义将在后面介绍。

2．存放在栈中的寄存器结构 pt_regs

在内核中，很多函数的参数是 pt_regs 数据结构，定义在 include/i386/ptrace.h 中：

```
struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
};
```

把这个结构与内核栈的内容相比较，会发现堆栈的内容是这个数据结构的一个映像。

3．保存现场的宏 SAVE_ALL

在中断发生前夕，要把所有相关寄存器的内容都保存在堆栈中,这是通过 SAVE_ALL 宏完成的：

```
#define SAVE_ALL \
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $ (__KERNEL_DS),%edx; \
        movl %edx,%ds; \
        movl %edx,%es;
```

该宏执行以后,堆栈内容如图 3.7 所示。把这个宏与图 3.6 结合起来就很容易理解图 3.7，在此对该宏再给予解释：

- CPU 在进入中断处理程序时自动将用户栈指针（如果更换堆栈）、EFLAGS 寄存器及返回地址一同压入堆栈。
- 段寄存器 DS 和 ES 原来的内容入栈，然后装入内核数据段描述符__KERNEL_DS（定义为 0x18），内核段的 DPL 为 0。

4．恢复现场的宏 RESTORE_ALL

当从中断返回时，恢复相关寄存器的内容，这是通过 RESTORE_ALL 宏完成的：

```
    #define RESTORE_ALL     \
        popl %ebx;      \
        popl %ecx;      \
         popl %edx;      \
         popl %esi;      \
         popl %edi;      \
         popl %ebp;      \
         popl %eax;      \
  1:      popl %ds;       \
  2:      popl %es;       \
          addl $4,%esp;   \
  3:      iret;
```

可以看出，RESTORE_ALL 与 SAVE_ALL 遥相呼应。当执行到 iret 指令时，内核栈又恢复到刚进入中断门时的状态，并使 CPU 从中断返回。

5．将当前进程的 task_struct 结构的地址放在寄存器中

```
 #define GET_CURRENT (reg)  \
         movl $-8192, reg; \
         andl %esp, reg
```

从下一章“task_struct 结构在内存存放”一节我们将知道，当前进程的 task_struct 存放在内核栈的底部，因此，以上两条指令就可以把 task_struct 结构的地址放在 reg 寄存器中。

3.4.4 中断处理程序的执行

从前面的介绍，我们已经知道了 i386 的中断机制及有关的初始化工作。现在，我们可以从中断请求的发生到 CPU 的响应，再到中断处理程序的调用和返回，沿着这一思路走一遍，以体会 Linux 内核对中断的响应及处理。

假定外设的驱动程序都已完成了初始化工作，并且已把相应的中断服务例程挂入到特定的中断请求队列。又假定当前进程正在用户空间运行（随时可以接受中断），且外设已产生了一次中断请求。当这个中断请求通过中断控制器 8259A 到达 CPU 的中断请求引线 INTR 时（参看图 3.1），CPU 就在执行完当前指令后来响应该中断。

CPU 从中断控制器的一个端口取得中断向量 I，然后根据 I 从中断描述符表 IDT 中找到相应的表项，也就是找到相应的中断门。因为这是外部中断，不需要进行“门级”检查，CPU 就可以从这个中断门获得中断处理程序的入口地址，假定为 IRQ0x05_interrupt。因为这里假定中断发生时 CPU 运行在用户空间（CPL＝3），而中断处理程序属于内核（DPL＝0），因此，要进行堆栈的切换。也就是说，CPU 从 TSS 中取出内核栈指针，并切换到内核栈（此时栈还为空）。当 CPU 进入 IRQ0x05_interrupt 时，内核栈如图 3.6 的①，栈中除用户栈指针、EFLAGS 的内容以及返回地址外再无其他内容。另外，由于 CPU 进入的是中断门（而不是陷阱门），因此，这条中断线已被禁用，直到重新启用。

我们用 IRQn_interrupt 来表示从 IRQ0x01_interrupt 到 IRQ0x0f_interrupt 任意一个中断处理程序。这个中断处理程序实际上要调用 do_IRQ()，而 do_IRQ()要调用 handle_IRQ_event()函数；最后这个函数才真正地执行中断服务例程（ISR）。图 3.8 给出了它们的调用关系。

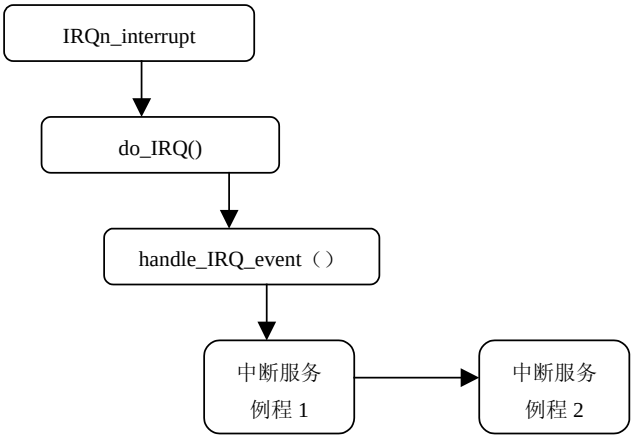


图 3.8　中断处理函数的调用关系

1．中断处理程序 IRQn_interrupt

我们首先看一下从 IRQ0x01_interrupt 到 IRQ0x0f_interrupt 的这 16 个函数是如何定义的，在 i8259.c 中定义了如下宏：

```
#define BI（x,y） \
        BUILD_IRQ（x##y）

#define BUILD_16_IRQS（x） \
        BI（x,0） BI（x,1） BI（x,2） BI（x,3） \
        BI（x,4） BI（x,5） BI（x,6） BI（x,7） \
        BI（x,8） BI（x,9） BI（x,a） BI（x,b） \
        BI（x,c） BI（x,d） BI（x,e） BI（x,f）

BUILD_16_IRQS（0x0）
```

经过 gcc 的预处理，宏定义 BUILD_16_IRQS（0x0） 会被展开成 BUILD_IRQ（0x00）至 BUILD_IRQ（0x0f）。BUILD_IRQ 宏是一段嵌入式汇编代码（在/include/i386/hw_irq.h 中），为了有助于理解，我们把它展开成下面的汇编语言片段：

```
IRQn_interrupt:
        pushl $n-256
jmp common_interrupt
```

把中断号减 256 的结果保存在栈中，这就是进入中断处理程序后第一个压入堆栈的值，也就是堆栈中 ORIG_EAX 的值，如图 3.7。这是一个负数，正数留给系统调用使用。对于每个中断处理程序，唯一不同的就是压入栈中的这个数。然后，所有的中断处理程序都跳到一段相同的代码 common_interrupt。这段代码可以在 BUILD_COMMON_IRQ 宏中找到，同样，我们略去其嵌入式汇编源代码，而把这个宏展开成下列的汇编语言片段：

```
common_interrupt:
        SAVE_ALL
```

```
        call do_IRQ
    jmp ret_from_intr
```

SAVE_ALL 宏已经在前面介绍过，它把中断处理程序会使用的所有 CPU 寄存器都保存在栈中。然后，BUILD_COMMON_IRQ 宏调用 do_IRQ（ ）函数，因为通过 CALL 调用这个函数，因此，该函数的返回地址被压入栈。当执行完 do_IRQ（ ），就跳转到 ret_from_intr（ ）地址（参见后面的“从中断和异常返回”）。

2. do_IRQ（ ）函数

do_IRQ()这个函数处理所有外设的中断请求。当这个函数执行时，内核栈从栈顶到栈底包括：

- do_IRQ（ ）的返回地址；
- 由 SAVE_ALL 推进栈中的一组寄存器的值；
- ORIG_EAX（即 n-256）；
- CPU 自动保存的寄存器。

该函数的实现用到中断线的状态，下面给予具体说明：

```
  #define IRQ_INPROGRESS  1    /* 正在执行这个 IRQ 的一个处理程序*/
  #define IRQ_DISABLED    2    /* 由设备驱动程序已经禁用了这条 IRQ 中断线 */
  #define IRQ_PENDING     4    /* 一个 IRQ 已经出现在中断线上，且被应答，但还没有为它提供服务 */
  #define IRQ_REPLAY      8    /* 当 Linux 重新发送一个已被删除的 IRQ 时 */
  #define IRQ_AUTODETECT  16   /* 当进行硬件设备探测时，内核使用这条 IRQ 中断线 */
  #define IRQ_WAITING     32   /*当对硬件设备进行探测时，设置这个状态以标记正在被测试的 irq */
  #define IRQ_LEVEL       64    /* IRQ level triggered */
  #define IRQ_MASKED      128    /* IRQ masked - shouldn't be seen again */
  #define IRQ_PER_CPU     256     /* IRQ is per CPU */
```

这 9 个状态的前 6 个状态比较常用，因此我们给出了具体解释。另外，我们还看到每个状态的常量是 2 的幂次方。最大值为 256（2^8），因此可以用一个字节来表示这 9 个状态，其中每一位对应一个状态。

该函数在 arch / /i386/kernel/irq.c 中定义如下：

```
asmlinkage unsigned int do_IRQ (struct pt_regs regs)
{
        /*  函数返回 0 则意味着这个 irq 正在由另一个 CPU 进行处理，
或这条中断线被禁用*/
        int irq = regs.orig_eax & 0xff;     /* 还原中断号 */
        int cpu = smp_processor_id();     /*获得 CPU 号*/
        irq_desc_t *desc = irq_desc + irq; /*在 irq_desc[]数组中获得 irq 的描述符*/
        struct irqaction * action;
        unsigned int status;

        kstat.irqs[cpu][irq]++;
```

```
        spin_lock (&desc->lock) ;   / *针对多处理机加锁* /
        desc->handler->ack (irq) ;   / *CPU 对中断请求给予确认* /

        status = desc->status & ~ (IRQ_REPLAY | IRQ_WAITING) ;
        status |= IRQ_PENDING; /* we _want_ to handle it */


        action = NULL;
        if  (! (status &  (IRQ_DISABLED | IRQ_INPROGRESS) ) )  {
                action = desc->action;
                status &= ~IRQ_PENDING; /* we commit to handling */
                status |= IRQ_INPROGRESS; /* we are handling it */
        }
        desc->status = status;
        if  (!action)
                goto out;
        for  (;;)  {
                 spin_unlock (&desc->lock) ;   /*进入临界区*
                 handle_IRQ_event (irq, &regs, action) ;
                 spin_lock (&desc->lock) ;    /*出临界区* /

                 if  (! (desc->status & IRQ_PENDING) )
                         break;
                 desc->status &= ~IRQ_PENDING;
        }
         desc->status &= ~IRQ_INPROGRESS;
out:
         /*
          * The ->end() handler has to deal with interrupts which got
         * disabled while the handler was running.
          */
         desc->handler->end (irq) ;
         spin_unlock (&desc->lock) ;

         if  (softirq_pending (cpu) )
                 do_softirq();    /*处理软中断* /
         return 1;
}
```

下面对这个函数进行进一步的讨论。

- 当执行到 for （;;）这个无限循环时，就准备对中断请求队列进行处理，这是由 handle_IRQ_event ()函数完成的。因为中断请求队列为一临界资源，因此在进入这个函数前要加锁。
- handle_IRQ_event ()函数的主要代码片段为：

```
if  (! (action->flags & SA_INTERRUPT) )
                __sti();    /*关中断*/
do {
               status |= action->flags;
                 action->handler (irq, action->dev_id, regs) ;
                 action = action->next;
         } while  (action) ;
```

```
    __cli();      /*开中断*/
```

这个循环依次调用请求队列中的每个中断服务例程。中断服务例程及其参数已经在前面进行过简单描述，至于更具体的解释将在驱动程序一章进行描述。

- 这里要说明的是，中断服务例程都在关中断的条件下进行（不包括非屏蔽中断），这也是为什么 CPU 在穿过中断门时自动关闭中断的原因。但是，关中断时间绝不能太长，否则就可能丢失其他重要的中断。也就是说，中断服务例程应该处理最紧急的事情，而把剩下的事情交给另外一部分来处理。即后半部分（bottom half）来处理，这一部分内容将在下一节进行讨论。
- 经验表明，应该避免在同一条中断线上的中断嵌套，内核通过 IRQ_PENDING 标志位的应用保证了这一点。当 do_IRQ()执行到 for （;;）循环时，desc->status 中的 IRQ_PENDING 的标志位肯定为 0（想想为什么？）。当 CPU 执行完 handle_IRQ_event ()函数返回时，如果这个标志位仍然为 0，那么循环就此结束。如果这个标志位变为 1，那就说明这条中断线上又有中断产生（对单 CPU 而言），所以循环又执行一次。通过这种循环方式，就把可能发生在同一中断线上的嵌套循环化解为“串行”。
- 不同的 CPU 不允许并发地进入同一中断服务例程，否则，那就要求所有的中断服务例程必须是“可重入”的纯代码。可重入代码的设计和实现就复杂多了，因此，Linux 在设计内核时巧妙地“避难就易”，以解决问题为主要目标。
- 在循环结束后调用 desc->handler->end()函数，具体来说，如果没有设置 IRQ_DISABLED 标志位，就调用低级函数 enable_8259A_irq()来启用这条中断线。
- 如果这个中断有后半部分,就调用 do_softirq()执行后半部分。

3.4.5 从中断返回

从前面的讨论我们知道，do_IRQ()这个函数处理所有外设的中断请求。这个函数执行的时候，内核栈栈顶包含的就是 do_IRQ()的返回地址，这个地址指向 ret_from_intr。实际上，ret_from_intr 是一段汇编语言的入口点，为了描述简单起见，我们以函数的形式提及它。虽然我们这里讨论的是中断的返回，但实际上中断、异常及系统调用的返回是放在一起实现的，因此，我们常常以函数的形式提到下面这 3 个入口点。

ret_from_intr()

终止中断处理程序。

ret_from_sys_call ( )

终止系统调用，即由 0x80 引起的异常。

ret_from_exception ( )

终止除了 0x80 的所有异常。

在相关的计算机课程中，我们已经知道从中断返回时 CPU 要做的事情，下面我们来看一下 Linux 内核的具体实现代码（在 entry.S 中）：

```
ENTRY (ret_from_intr)
        GET_CURRENT (%ebx)
ret_from_exception:
         movl EFLAGS (%esp),%eax          # mix EFLAGS and CS
         movb CS (%esp),%al
```

```
        testl $(VM_MASK | 3),%eax       # return to VM86 mode or non-supervisor?
        jne ret_from_sys_call
        jmp restore_all
```

这里的 GET_CURRENT（%ebx）将当前进程 task_struct 结构的指针放入寄存器 EBX 中，此时，内核栈的内容还如图 3.7 所示。然后两条“mov”指令是为了把中断发生前夕 EFALGS 寄存器的高 16 位与代码段 CS 寄存器的内容拼接成 32 位的长整数，其目的是要检验：

- 中断前夕 CPU 是否够运行于 VM86 模式；
- 中断前夕 CPU 是运行在用户空间还是内核空间。

VM86 模式是为在 i386 保护模式下模拟运行 DOS 软件而设置的，EFALGS 寄存器高 16 位中有个标志位表示 CPU 是否运行在 VM86 模式，我们在此不予详细讨论。CS 的最低两位表示中断发生时 CPU 的运行级别 CPL，若这两位为 3，说明中断发生于用户空间。

如果中断发生在内核空间，则控制权直接转移到 restore_all。如果中断发生于用户空间（或 VM86 模式），则转移到 ret_from_sys_call：

```
ENTRY(ret_from_sys_call)
        cli                             # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
restore_all:
        RESTORE_ALL

  reschedule:
        call SYMBOL_NAME(schedule)     # test
        jmp ret_from_sys_call
```

进入 ret_from_sys_call 后，首先关中断，也就是说，执行这段代码时 CPU 不接受任何中断请求。然后，看调度标志是否为非 0，其中常量 need_resched 定义为 20，need_resched（%ebx）表示当前进程 task_struct 结构中偏移量 need_resched 处的内容，如果调度标志为非 0，说明需要进行调度，则去调用 schedule()函数进行进程调度，这将在第五章进行详细讨论。

同样，如果当前进程的 task_struct 结构中的 sigpending 标志为非 0，则表示该进程有信号等待处理，要先处理完这些信号后才从中断返回，关于信号的处理将在“进程间通信”一章进行讨论。处理完信号，控制权还是返回到 restore_all。RESTORE_ALL 宏操作在前面已经介绍过，也就是恢复中断现场，彻底从中断返回。

3.5　中断的后半部分处理机制

从上面的讨论我们知道，Linux 并不是一口气把中断所要求的事情全部做完，而是分两部分来做，本节我们将具体描述内核怎样处理中断的后半部分。

3.5.1 为什么把中断分为两部分来处理

中断服务例程一般都是在中断请求关闭的条件下执行的,以避免嵌套而使中断控制复杂化。但是，中断是一个随机事件，它随时会到来，如果关中断的时间太长，CPU 就不能及时响应其他的中断请求，从而造成中断的丢失。因此，内核的目标就是尽可能快地处理完中断请求，尽其所能把更多的处理向后推迟。例如，假设一个数据块已经达到了网线，当中断控制器接受到这个中断请求信号时，Linux 内核只是简单地标志数据到来了，然后让处理器恢复到它以前运行的状态，其余的处理稍后再进行（如把数据移入一个缓冲区，接受数据的进程就可以在缓冲区找到数据）。因此，内核把中断处理分为两部分：前半部分（top half）和后半部分（bottom half），前半部分内核立即执行，而后半部分留着稍后处理，如图 3.9 所示。

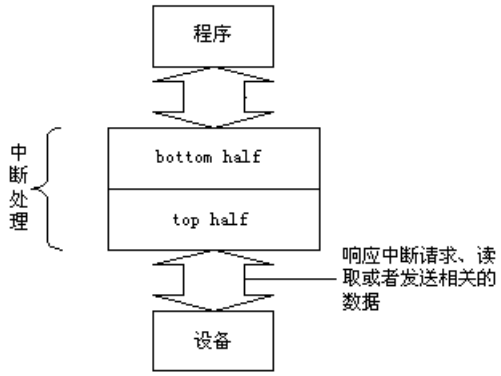


图 3.9 中断的分割

首先，一个快速的“前半部分”来处理硬件发出的请求，它必须在一个新的中断产生之前终止。通常情况下，除了在设备和一些内存缓冲区（如果设备用到了 DMA，就不止这些）之间移动或传送数据，确定硬件是否处于健全的状态之外，这一部分做的工作很少。

然后，就让一些与中断处理相关的有限个函数作为“后半部分”来运行：

- 允许一个普通的内核函数，而不仅仅是服务于中断的一个函数，能以后半部分的身份来运行；
- 允许几个内核函数合在一起作为一个后半部分来运行。

后半部分运行时是允许中断请求的，而前半部分运行时是关中断的，这是二者之间的主要区别。

3.5.2 实现机制

Linux 内核为将中断服务分为两部分提供了方便，并设立了相应的机制。在以前的内核中，这个机制就叫 bottom half（简称 bh），但在 2.4 版中有了新的发展和推广，叫做软中断（softirq）机制。

1．bh 机制

以前内核中的 bh 机制设置了一个函数指针数组 bh_base[]，它把所有的后半部分都组织起来，其大小为 32，数组中的每一项就是一个后半部分，即一个 bh 函数。同时，又设置了两个 32 位无符号整数 bh_active 和 bh_mask，每个无符号整数中的一位对应着 bh_base[]中的一个元素，如图 3.10 所示。

在 2.4 以前的内核中，每次执行完 do_IRQ()中的中断服务例程以后，以及每次系统调用结束之前，就在一个叫 do_bottom_half()的函数中执行相应的 bh 函数。

在 do_bottom_half()中对 bh 函数的执行是在关中断的情况下进行的，也就是说对 bh 的执行进行了严格的“串行化”，这种方式简化了 bh 的设计，这是因为，对单 CPU 来说，bh 函数的执行可以不嵌套；而对于多 CPU 来说，在同一时间内最多只允许一个 CPU 执行 bh 函数。

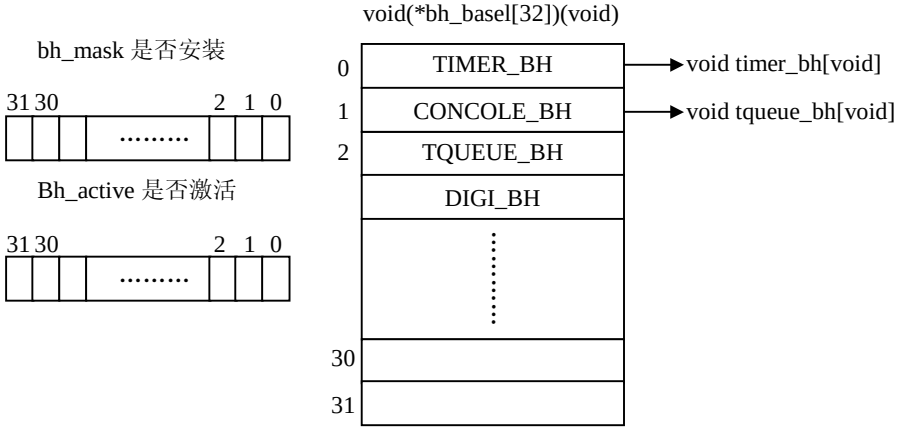


图 3.10　bh 机制示意图

这种简化了的设计在一定程度上保证了从单 CPU 到多 CPU SMP 结构的平稳过渡，但随着时间的推移，就会发现这样的处理对于 SMP 的性能有不利的影响。因为，当系统中有很多个 bh 函数需要执行时，bh 函数的“串行化”却只能使一个 CPU 执行一个 bh 函数，其他 CPU 即使空闲，也不能执行其他的 bh 函数。由此可以看出，bh 函数的串行化是针对所有 CPU 的，根本发挥不出多 CPU 的优势。

那么，在新内核的设计中，是改进 bh 机制还是抛弃 bh 机制，建立一种新的机制？2.4 选择了一种折衷的办法，继续保留 bh 机制，另外增加一种或几种机制，并把它们纳入一个统一的框架中，这就是 2.4 内核中的软中断（softirq）机制。

2．软中断机制

软中断机制也是推迟内核函数的执行，然而，与 bh 函数严格地串行执行相比，软中断却在任何时候都不需要串行化。同一个软中断的两个实例完全有可能在两个 CPU 上同时运行。当然，在这种情况下，软中断必须是可重入的。软中断给网络部分带来的好处尤为突出，因为 2.4 内核中用两个软中断代替原来的一个 NET_BH 函数，这就使得在多处理机系统上软中断的执行更为高效。

3．Tasklet 机制

另一个类似于 bh 的机制叫做 tasklet。Tasklet 建立在软中断之上，但与软中断的区别是，同一个 tasklet 只能运行在一个 CPU 上，而不同的 tasklet 可以同时运行在不同的 CPU 上。在这种情况下，tasklet 就不需要是可重入的，因此，编写 tasklet 比编写一个软中断要容易。

Bh 机制在 2.4 中依然存在，但不是作为一个单独的机制存在，而是建立在 tasklet 之上。因此，在 2.4 版中，设备驱动程序的开发者必须更新他们原来的驱动程序，用 tasklet 代替 bh。

3.5.3 数据结构的定义

在具体介绍软中断处理机制之前，我们先介绍一下相关的数据结构，这些数据结构大部分都在 / include/linux/interrupt.h 中。

1．与软中断相关的数据结构

软中断本身是一种机制，同时也是一种基本框架。在这个框架中，既包含了 bh 机制，也包含了 tasklet 机制。

（1）内核定义的软中断

```
enum
        {
         HI_SOFTIRQ=0,
         NET_TX_SOFTIRQ,
         NET_RX_SOFTIRQ,
         TASKLET_SOFTIRQ
         };
```

内核中用枚举类型定义了 4 种类型的软中断，其中 NET_TX_SOFTIRQ 和 NET_RX_SOFTIRQ 两个软中断是专为网络操作而设计的，而 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 是针对 bh 和 tasklet 而设计的软中断。编码作者在源码注释中曾提到，一般情况下，不要再分配新的软中断。

（2）软中断向量

```
      struct softirq_action
{
                 void     (*action) (struct softirq_action *) ;
      void    *data;
         }

     static struct softirq_action softirq_vec[32] __cacheline_aligned;
```

从定义可以看出，内核定义了 32 个软中断向量，每个向量指向一个函数，但实际上，内核目前只定义了上面的 4 个软中断，而我们后面主要用到的为 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 两个软中断。

（3）软中断控制 / 状态结构

softirq_vec [] 是个全局量，系统中每个 CPU 所看到的是同一个数组。但是，每个 CPU

各有其自己的“软中断控制/状态”结构，这些数据结构形成一个以CPU编号为下标的数组irq_stat[]（定义在include/i386/hardirq.h中）

```
typedef struct {
        unsigned int __softirq_pending;
        unsigned int __local_irq_count;
        unsigned int __local_bh_count;
        unsigned int __syscall_count;
        struct task_struct * __ksoftirqd_task; /* waitqueue is too large */
        unsigned int __nmi_count;       /* arch dependent */
} ____cacheline_aligned irq_cpustat_t;

irq_cpustat_t irq_stat[NR_CPUS];
```

irq_stat[]数组也是一个全局量，但是各个CPU可以按其自身的编号访问相应的域。于是，内核定义了如下宏（在include/linux/irq_cpustat.h中）：

```
#ifdef CONFIG_SMP
#define __IRQ_STAT(cpu, member)   (irq_stat[cpu].member)
#else
#define __IRQ_STAT(cpu, member)   ((void)(cpu), irq_stat[0].member)
#endif

/* arch independent irq_stat fields */
#define softirq_pending(cpu)     __IRQ_STAT((cpu), __softirq_pending)
#define local_irq_count(cpu)     __IRQ_STAT((cpu), __local_irq_count)
#define local_bh_count(cpu)      __IRQ_STAT((cpu), __local_bh_count)
#define syscall_count(cpu)       __IRQ_STAT((cpu), __syscall_count)
#define ksoftirqd_task(cpu)      __IRQ_STAT((cpu), __ksoftirqd_task)
   /* arch dependent irq_stat fields */
#define nmi_count(cpu)           __IRQ_STAT((cpu), __nmi_count)          /* i386, ia64
*/
```

2．与tasklet相关的数据结构

与bh函数相比，tasklet是“多序”的bh函数。内核中用tasklet_task来定义一个tasklet：

```
 struct tasklet_struct
 {
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void (*func)(unsigned long);
        unsigned long data;
};
```

从定义可以看出，tasklet_struct是一个链表结构，结构中的函数指针func指向其服务程序。内核中还定义了一个以CPU编号为下标的数组tasklet_vec[]和tasklet_hi_vec[]：

```
   struct tasklet_head
{
        struct tasklet_struct *list;
} __attribute__ ((__aligned__(SMP_CACHE_BYTES)));
```

```
extern struct tasklet_head tasklet_vec[NR_CPUS];
extern struct tasklet_head tasklet_hi_vec[NR_CPUS];
```

这两个数组都是 tasklet_head 结构数组，每个 tasklet_head 结构就是一个 tasklet_struct 结构的队列头。

3．与 bh 相关的数据结构

前面我们提到，bh 建立在 tasklet 之上，更具体地说，对一个 bh 的描述也是 tasklet_struct 结构，只不过执行机制有所不同。因为在不同的 CPU 上可以同时执行不同的 tasklet，而任何时刻，即使在多个 CPU 上，也只能有一个 bh 函数执行。

（1）bh 的类型

```
enum {
        TIMER_BH = 0,   /* 定时器 */
        TQUEUE_BH,      /* 周期性任务队列 */
        DIGI_BH,        /* DigiBoard PC/Xe */
        SERIAL_BH,      /* 串行接口 */
        RISCOM8_BH,          /* RISCom/8 */
        SPECIALIX_BH,   /* Specialix IO8+ */
        AURORA_BH,           /* Aurora 多端口卡（SPARC）*/
        ESP_BH,              /* Hayes ESP 串行卡 */
        SCSI_BH,        /* SCSI 接口*/
        IMMEDIATE_BH,   /* 立即任务队列*/
        CYCLADES_BH,    /* Cyclades Cyclom-Y 串行多端口 */
        CM206_BH,       /* CD-ROM Philips/LMS cm206 磁盘 */
        JS_BH,               /* 游戏杆（PC IBM）*/
        MACSERIAL_BH,   /* Power Macintosh 的串行端口 */
        ISICOM_BH       /* MultiTech 的 ISI 卡*/
};
```

在给出 bh 定义的同时，我们也给出了解释。从定义中可以看出，有些 bh 与硬件设备相关，但这些硬件设备未必装在系统中，或者仅仅是针对 IBM PC 兼容机之外的某些平台。

（2）bh 的组织结构

在 2.4 以前的版本中，把所有的 bh 用一个 bh_base[]数组组织在一起，数组的每个元素指向一个 bh 函数：

```
static void （*bh_base[32]）（void）;
```

2.4 版中保留了上面这种定义形式，但又定义了另外一种形式：

```
struct tasklet_struct bh_task_vec[32];
```

这也是一个有 32 个元素的数组，但数组的每个元素是一个 tasklet_struct 结构，数组的下标就是上面定义的枚举类型中的序号。

3.5.4　软中断、bh 及 tasklet 的初始化

1．Tasklet 的初始化

Tasklet 的初始化是由 tasklet_ init()函数完成的：

```
 void tasklet_init（struct tasklet_struct *t,
```

```
                    void (*func) (unsigned long), unsigned long data)
{
        t->next = NULL;
        t->state = 0;
        atomic_set (&t->count, 0) ;
        t->func = func;
        t->data = data;
}
```

其中，atomic_set()为原子操作,它把 t->count 置为 0。

2．软中断的初始化

首先通过 open_softirq()函数打开软中断：

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

然后，通过 softirq_init()函数对软中断进行初始化：

```
void __init softirq_init()
{
        int i;

        for (i=0; i<32; i++)
              tasklet_init (bh_task_vec+i, bh_action, i) ;

        open_softirq (TASKLET_SOFTIRQ, tasklet_action, NULL) ;
        open_softirq (HI_SOFTIRQ, tasklet_hi_action, NULL) ;
}
```

对于 bh 的 32 个 tasklet_struct，调用 tasklet_init 以后，它们的函数指针 func 全部指向 bh_action()函数，也就是建立了 bh 的执行机制，但具体的 bh 函数还没有与之挂勾，就像具体的中断服务例程还没有挂入中断服务队列一样。同样，调用 open_softirq()以后，软中断 TASKLET_SOFTIRQ 的服务例程为 tasklet_action()，而软中断 HI_SOFTIRQ 的服务例程为 tasklet_hi_action()。

3．Bh 的初始化

bh 的初始化是由 init_bh()完成的：

```
void init_bh (int nr, void (*routine) (void) )
{
        bh_base[nr] = routine;
        mb();
 }
```

这里调用的函数 mb()与 CPU 中执行指令的流水线有关，我们对此不进行进一步讨论。下面看一下几个具体 bh 的初始化（在 kernel/sched.c 中）：

```
init_bh (TIMER_BH,timer_bh) ;
init_bh (TUEUE_BH,tqueue_bh) ;
init_bh (IMMEDIATE_BH,immediate_bh) ;
```

初始化以后，bh_base[TIMER_BH]处理定时器队列 timer_bh，每个时钟中断都会激活 TIMER_BH,在第五章将会看到,这意味着大约每隔 10ms 这个队列运行一次。bh_base[TUEUE_BH]处理周期性的任务队列 tqueue_bh，而 bh_base[IMMEDIATE_BH]通常被驱动程序所调用，请求某个设备服务的内核函数可以链接到 IMMEDIATE_BH 所管理的队列 immediate_bh 中，在该队列中排队等待。

3.5.5 后半部分的执行

1. Bh 的处理

当需要执行一个特定的 bh 函数（例如 bh_base[TIMER_BH]()）时，首先要提出请求，这是由 mark_bh()函数完成的（在 Interrupt.h 中）：

```
static inline void mark_bh (int nr)
{
        tasklet_hi_schedule (bh_task_vec+nr) ;
}
```

从上面的介绍我们已经知道，bh_task_vec[]每个元素为 tasklet_struct 结构，函数的指针 func 指向 bh_action()。

接下来，我们来看 tasklet_hi_schedule()完成什么功能：

```
static inline void tasklet_hi_schedule (struct tasklet_struct *t)
{
        if  (!test_and_set_bit (TASKLET_STATE_SCHED, &t->state) )
        int cpu = smp_processor_id();
        unsigned long flags;

       local_irq_save (flags) ;
        t->next = tasklet_hi_vec[cpu].list;
        tasklet_hi_vec[cpu].list = t;
        cpu_raise_softirq (cpu, HI_SOFTIRQ) ;
        local_irq_restore (flags) ;
}
```

其中 smp_processor_id()返回当前进程所在的 CPU 号，然后以此为下标从 tasklet_hi_vec[]中找到该 CPU 的队列头，把参数 t 所指向的 tasklet_struct 结构链入这个队列。由此可见，当某个 bh 函数被请求执行时，当前进程在哪个 CPU 上，就把这个 bh 函数“调度”到哪个 CPU 上执行。另一方面，tasklet_struct 代表着将要对 bh 函数的一次执行，在同一时间内，只能把它链入一个队列中，而不可能同时出现在多个队列中。对同一个 tasklet_struct 结构，如果已经对其调用了 tasklet_hi_schedule()函数，而尚未得到执行，就不允许再将其链入该队列，所以标志位 TASKLET_STATE_SCHED 就是保证这一点的。最后，通过 cpu_raise_softirq()发出软中断请求，其中的参数 HI_SOFTIRQ 表示 bh 与 HI_SOFTIRQ 软中断对应。

软中断 HI_SOFTIRQ 的服务例程为 tasklet_hi_action()：

```
static void tasklet_hi_action (struct softirq_action *a)
{
```

```
        int cpu = smp_processor_id();
        struct tasklet_struct *list;

        local_irq_disable();
        list = tasklet_hi_vec[cpu].list;
        tasklet_hi_vec[cpu].list = NULL;        /*临界区加锁*/
        local_irq_enable();

       while (list) {
               struct tasklet_struct *t = list;

               list = list->next;

               if (tasklet_trylock(t)) {
                       if (!atomic_read(&t->count)) {
                              if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                                      BUG();
                              t->func(t->data);
                             tasklet_unlock(t);
                             continue;
                       }
                       tasklet_unlock(t);
               }

              local_irq_disable();
               t->next = tasklet_hi_vec[cpu].list;
              tasklet_hi_vec[cpu].list = t;
               __cpu_raise_softirq(cpu, HI_SOFTIRQ);
               local_irq_enable();
        }
}
```

这个函数除了加锁机制以外，读起来比较容易。其中要说明的是 t->func（t->data）语句，这条语句实际上就是调用 bh_action()函数：

```
/* BHs are serialized by spinlock global_bh_lock.

   It is still possible to make synchronize_bh() as
   spin_unlock_wait(&global_bh_lock). This operation is not used
  by kernel now, so that this lock is not made private only
   due to wait_on_irq().

   It can be removed only after auditing all the BHs.
 */
spinlock_t global_bh_lock = SPIN_LOCK_UNLOCKED;

static void bh_action(unsigned long nr)
{
        int cpu = smp_processor_id();

        if (!spin_trylock(&global_bh_lock))
                goto resched;

        if (!hardirq_trylock(cpu))
```

```
                goto resched_unlock;

        if (bh_base[nr])
                bh_base[nr]();

        hardirq_endlock(cpu);
        spin_unlock(&global_bh_lock);
       return;

resched_unlock:
        spin_unlock(&global_bh_lock);
resched:
        mark_bh(nr);
}
```

这里对 bh 函数的执行又设置了两道锁。一是 hardirq_trylock()，这是防止从一个硬中断内部调用 bh_action()。另一道锁是 spin_trylock()。这把锁就是全局量 global_bh_lock，只要有一个 CPU 在这个锁所锁住的临界区运行，别的 CPU 就不能进入这个区间，所以在任何时候最多只有一个 CPU 在执行 bh 函数。至于根据 bh 函数的编号执行相应的函数，那就比较容易理解了。

2．软中断的执行

内核每当在 do_IRQ()中执行完一个中断请求队列中的中断服务例程以后，都要检查是否有软中断请求在等待执行。下面是 do_IRQ()中的一条语句：

```
if (softirq_pending(cpu))
            do_softirq();
```

在检测到软中断请求以后，就要通过 do_softirq()执行软中断服务例程，其代码在/kernel/softirq.c 中：

```
asmlinkage void do_softirq()
{
        int cpu = smp_processor_id();
        __u32 pending;
        long flags;
        __u32 mask;

       if (in_interrupt())
                return;

        local_irq_save(flags);/*把 eflags 寄存器的内容保存在 flags 变量中*/

        pending = softirq_pending(cpu);

       if (pending) {
               struct softirq_action *h;

               mask = ~pending;
               local_bh_disable();
restart:
               /* Reset the pending bitmask before enabling irqs */
```

```
                softirq_pending (cpu)  = 0;

                local_irq_enable();   /*开中断*/

                h = softirq_vec;

                do {
                        if  (pending & 1)
                                h->action (h) ;
                         h++;
                         pending >>= 1;
                } while  (pending) ;

                local_irq_disable();  / *关中断*/

                pending = softirq_pending (cpu) ;
                if  (pending & mask)  {
                        mask &= ~pending;
                         goto restart;
                }
                __local_bh_enable();

                if  (pending)
                        wakeup_softirqd (cpu) ;
        }

         local_irq_restore (flags) ;  / *恢复 eflags 寄存器的内容* /
    }
```

从 do_softirq()的代码可以看出，使 CPU 不能执行软中断服务例程的“关卡”只有一个，那就是 in_interrupt()，这个宏限制了软中断服务例程既不能在一个硬中断服务例程内部执行，也不能在一个软中断服务例程内部执行（即嵌套）。但这个函数并没有对中断服务例程的执行进行“串行化”限制。这也就是说，不同的 CPU 可以同时进入对软中断服务例程的执行，每个 CPU 分别执行各自所请求的软中断服务。从这个意义上说，软中断服务例程的执行是“并发的”、“多序的”。但是，这些软中断服务例程的设计和实现必须十分小心，不能让它们相互干扰（例如通过共享的全局变量）。

从前面对软中断数据结构的介绍可以知道，尽管内核最多可以处理 32 个软中断，但目前只定义了 4 个软中断。在对软中断进行初始化时，soft_Init()函数只初始化了两个软中断 TASKLET_SOFTIRQ 和 HI_SOFTIRQ，这两个软中断对应的服务例程为 tasklet_action()和 tasklet_hi_action()。因此，do_softirq()中的 do_while 循环实际上是调用这两个函数。前面已经给出了 tasklet_hi_action()的源代码，而 tasklet_action()的代码与其基本一样，在此不再给出。

3.5.6　把 bh 移植到 tasklet

在 Linux 2.2 中，对中断的后半部分处理只提供了 bh 机制，而在 2.4 中新增加了两种机制：软中断和 tasklet。通过上面的介绍我们知道，同一个软中断服务例程可以同时在不

同的 CPU 上运行。为了提高 SMP 的性能，软中断现在主要用在网络子系统中。多个 tasklet 可以在多个不同的 CPU 上运行，但一个 CPU 一次只能处理一个 tasklet。bh 由内核进行了串行化处理，也就是在 SPM 环境中，某一时刻，一个 bh 函数只能由一个 CPU 来执行。如果要把 Linux 2.2 中的 bh 移植到 2.4 的 tasklet，请按下面方法进行。

1. Linux 2.4 中对 bh 的处理

假设一个 bh 为 FOO_BH（FOO 表示任意一个），其处理函数为 foo_bh，则：

（1）处理函数的原型为：void foo_bh（void）；

（2）通过 init_bh（FOO_BH, foo_bh）函数对 foo_bh 进行初始化；

（3）通过 mark_bh（FOO_BH）函数提出对 foo_bh()的执行请求。

2. 把 bh 移植到 tasklet

（1）处理函数的原型为：void foo_bh（unsigned long data）；

（2）通过宏 DECLARE_TASKLET_DISABLED（foo_tasklet, foo_bh, 0）或

```
struct tasklet_struct foo_tasklet;
tasklet_init（&foo_tasklet, foo_bh, 0）;
tasklet_disable（&foo_tasklet）;
```

对 foo_tasklet 进行初始化

（3）通过

tasklet_enable（&foo_tasklet）；

tasklet_schedule（&foo_tasklet）；

对 foo_tasklet 进行调度。