10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象
华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《Linux 内核修炼之道》

作者：华清远见

第 6 章 中断与中断处理

本章简介

李白在《望天门山》中的“天门中断楚江开，碧水东流至此回”是说由于楚江怒涛的冲击，才撞开了“天门”，使它中断而成为东西两山。

中断这个名字十分形象，因为它终止了系统正常的处理过程。在前面第 5 章中我们已经看到了中断的一个例子——提供系统调用基本机制的软件中断。在本章中，我们接着讲解硬件中断。

6.1 概述

管理系统中存在的各种设备是内核的主要职责，内核一般可以通过两种方式来履行这个职责。

轮询：内核以一定的周期访问设备，查询设备的状态并进行处理。

中断：设备在需要时通知内核，内核收到设备的请求后再做出相应处理。

使用轮询方式时，内核对设备的访问按照一定频率周期地进行，因而常常会出现设备的请求不能得到及时的处理，或者在内核一次又一次的访问中没有发现设备的任何请求，进而浪费了大量的 CPU 资源的情况。

而使用中断方式时则不同，内核成为被动的一方，只有在收到设备的中断请求时，才中断当前正在进行的工作去进行中断处理。

从物理上看，中断是硬件设备产生的一种电信号，通过中断控制器发送给 CPU，CPU 检测到该中断信号后再通知内核，然后由内核完成后续的处理。内核不再需要刻意按照一定的周期去访问设备，提高了 CPU 的利用率。

6.1.1 中断分类

中断可分为同步（synchronous）中断和异步（asynchronous）中断，如图 6.1 所示。

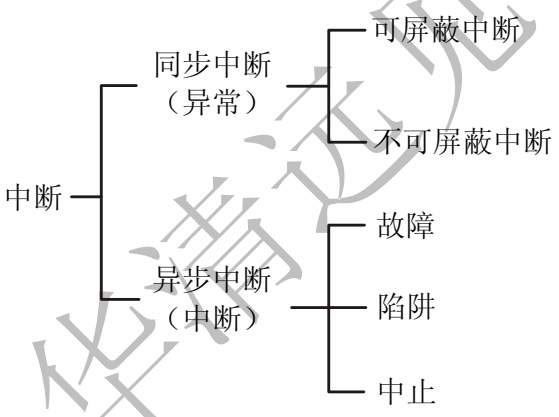

图 6.1 中断分类

同步中断由 CPU 本身产生，又称为内部中断。这里的同步是指中断请求信号与代码指令之间的同步执行，即只有在一条指令执行完毕后，而不是执行期间，CPU 才会发出中断。

异步中断由其他硬件设备产生，又称为外部中断。与同步中断相反，异步中断可于任何时间产生，包括指令执行期间。

通常将同步中断称为异常（exeption），将异步中断称为中断（interrupt），我们也将采用这种说法。

异常产生于 CPU 在指令执行期间检测到异常或非法的条件时，与当前正在执行的指令有直接的联系，比如，执行除法指令时，除数为 0。而中断通常用于指示设备的特定操作已经完成，与当前正执行的指令之间并没有直接联系。

异常包括故障（Fault）、陷阱（Trap）和中止（Abort）。故障是指在引起异常的指令之前，把该异常情况通知给系统的一种异常。故障处理结束后，返回到引起故障的指令并执行，比如被 0 除。

陷阱是指在引起异常的指令之后，把该异常情况通知给系统的一种异常。在转入相应的处理程序进行陷阱处理时，引起陷阱的指令应该已经正常完成，因此陷阱处理结束后，将返回到引起陷阱的指令的下一条指令并执行。

中止是在系统出现严重情况时，通知系统的一种异常。引起中止的指令是无法确定的，产生中止时，正执行的程序不能被恢复执行，比如硬件故障。

华清远见

中断包括可屏蔽中断（Maskable interrupt）和不可屏蔽中断（Nomaskable interrupt）。CPU 提供了两条外接引脚用于中断，即 NMI 和 INTR。其中，NMI 用于不可屏蔽中断，比如电源掉电，一旦产生，CPU 必须立即无条件响应，否则进行其他工作毫无意义，INTR 用于可屏蔽中断，主要是外部 I/O 设备的中断信号，比如打印机中断，这些中断信号需要通过中断控制器传递给 CPU。可屏蔽中断有屏蔽（masked）和非屏蔽（unmasked）两种状态，处于屏蔽状态的中断将被忽略。

可屏蔽中断受 EFLAGS 寄存器的中断允许标志位 IF 控制。

6.1.2 PIC vs APIC

中断控制器有可编程中断控制器(Programmable Interrupt Controller，PIC)和高级可编程中断控制器（Advanced Programmable Interrupt Controller，APIC）两种。

前者只能用于单处理器（Uni-processor，UP）平台，而后者则可以用于多处理器（Multiple Processor，MP）平台。目前，无论 UP 平台还是 MP 平台，多数使用的都是 APIC。

1．可编程中断控制器 8259A

单个 8259A 芯片只能管理 8 个中断源，若是需要管理更多的中断源，则必须将多个芯片级联。虽然最多可以级联成 64 级，但传统的 PIC 都是采用两个 8259A 级联的方式，支持 15 个中断源（第一个 8259A 芯片的 IR2 引脚用于串联第二个 8259A 芯片），如图 6.2 所示。

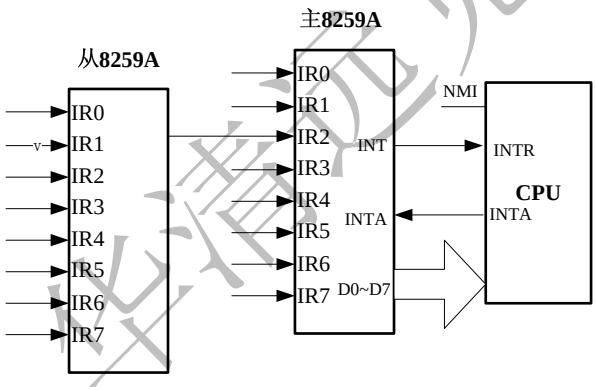

图 6.2 8259A 级联结构

中断控制器与其他硬件设备相连接的各条线被称做中断线。由于数目有限，所以中断线是非常宝贵的资源，使用之前，必须首先在设备的驱动程序中进行中断线的申请，获得该中断线的控制权，才能发送中断信号。

中断线又被称为 IRQ（Interrupt ReQuest）线，只有当设备确实需要使用中断的时候才能申请占用，或者在申请时采用与其他设备共享中断线的方式。

中断线按照一定的顺序进行编号，这个编号即是中断号。

8259A 的输出连接到 CPU 的 INTR 引脚上。外设发出中断信号时，若对应的中断没有被屏蔽，则拉高 INTR 引脚通知 CPU 有中断产生，CPU 可以通过 INTA 引脚应答，表示收到中断请求。

2．高级可编程中断控制器 APIC

与 8259A 相比，APIC 最大的优势是可适用于 MP 平台，以及可支持更多的中断源。

APIC 由两部分组成：本地高级中断控制器（Local APIC，LAPIC），位于 CPU 中，主要负责传递中断信号到指定的处理器，在 MP 平台，每个 CPU 都具有一个自己的 LAPIC；I/O 高级中断控制器（I/O APCI，IOAPIC）负责收集来自 I/O 设备的中断信号并分发给 LAPIC，系统中最多可以拥有 8 个 I/O APIC。

在一个典型的 MP 平台上，通常有一个 IOAPIC 和多个 LAPIC，所有的 LAPIC 都连接到 I/O APIC，形成一个多级的 APIC 中断分发网络，如图 6.3 所示。

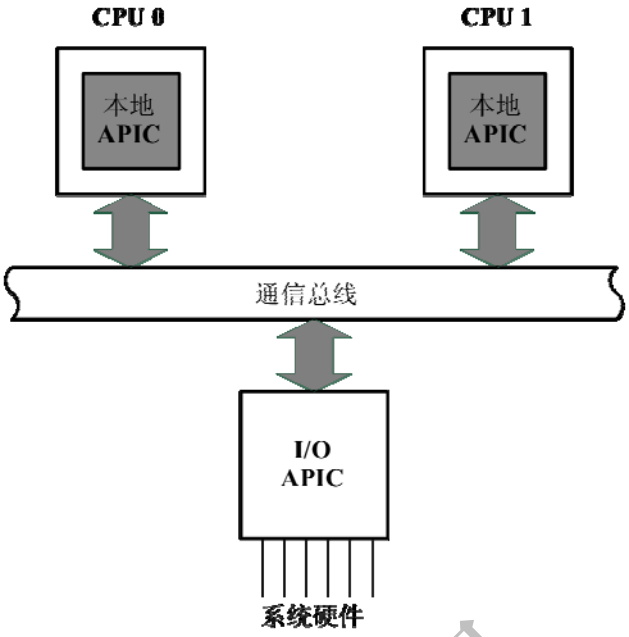

图 6.3　APIC 系统结构

可以通过查看/proc/interrupts 文件来获悉系统有没有在使用 APIC:

```
$ cat /proc/interrupts
          CPU0
  0:   20565961   IO-APIC-edge  timer
  1:       3763   IO-APIC-edge  i8042
  6:          4   IO-APIC-edge  floppy
  7:          0   IO-APIC-edge  parport0
  8:          1   IO-APIC-edge  rtc
  9:          0   IO-APIC-level  acpi
 12:       3095   IO-APIC-edge  i8042
 15:     737432   IO-APIC-edge  ide1
169:      11066   IO-APIC-level  eth0
177:          0   IO-APIC-level  uhci_hcd:usb1, es1371
185:      19441   IO-APIC-level  ioc0
NMI:          0
LOC:   19332194
ERR:          0
MIS:          0
```

如果看到显示有 IOAPIC，就表示系统当前在使用 APIC。

6.1.3　中断号 vs 中断向量

无论是中断还是异常，CPU 的响应过程基本一致，即根据中断源所提供的中断向量，从中断描述符表中获取相应处理程序的地址，然后执行。

Intel X86 能够支持 256 种不同的中断，并将它们从 0～255 进行编号，用于区分不同的中断源。这个 8 位的编号通常被称为中断向量。

与中断号不同，中断向量是从 CPU 角度看到的中断信号划分。当 I/O 设备把中断信号发送给中断控制器时，与之关联的是一个中断号；而当中断控制器将该中断信号传递给 CPU 时，与之关联的则是一个中断向量。

中断号与中断向量之间存在一对一的映射关系。对于 8259A，缺省是中断号加上 32 等于对应的中断向量。

异常与不可屏蔽中断的中断向量是固定的。Intel 将 0～31 之间的 32 个中断向量保留，用于处理异常和不可屏蔽中断。如表 6.1 所示即为 I386 平台的异常分布情况。

表 6.1　　I386 平台的异常分布

中断向量	名　称	类　型	描述或示例
0	除法错误	故障	被 0 除
1	调试	故障/陷阱	对代码进行逐步调试
2	不可屏蔽中断		利用 CPU NMI 引脚的那些中断，比如电源掉电
3	断点	陷阱	执行 int 3 指令时，可用于调试
4	溢出	陷阱	执行 into 指令时
5	边界检测	故障	bound 指令发现检测的值超出了给定的范围
6	非法操作码	故障	检测到无效的操作码
7	设备不可用	故障	用于支持协处理器
8	双重故障	中止	系统正在处理一个异常时，又检测到一个异常，如果处理器不能够串行进行处理，则产生双重故障异常
9	协处理器段越界	中止	浮点指令操作数超出段界限时
10	无效 TSS	故障	当正从任务状态段 TSS 装入段选择符时，如果发生段不存在异常（中断向量 11）之外的其他异常时，就产生无效 TSS 异常
11	段不存在	故障	引用一个不存在的内存段
12	堆栈段异常	故障	在堆栈操作时，超出段界限所规定的范围

续表

中断向量	名　称	类　型	描述或示例
13	通用保护异常	故障	其他的段异常
14	页异常	故障	寻址的页不在内存
15	保留		
16	协处理器出错	故障	协处理器发生了未被屏蔽的数字错误
17	对齐检查	故障	操作数的地址没有被正确地排列
18	机器检查	中止	检测到 CPU 或总线错误
19	SIMD 协处理器出错	故障	SIMD 协处理器遇到不能处理的指令
20～31	保留		

6.1.4　中断描述符表

在实模式下，从地址 0 开始的 1KB 大小内存构成一个中断向量表，表中每一项包括 4 个字节，对应一个中断向量，保存了该中断向量所对应中断处理程序的入口地址，如图 6.4 所示。

进入保护模式后，中断向量表改名为中断描述符表（Interrupt Descriptor Table，IDT），并允许在任意内存地址存放，因此 CPU 专门设置了一个中断描述符表寄存器 IDTR，用来存储 IDT 在内存中的起始地址。

同时 IDT 的每个表项被称为门描述符（Gate Descriptor），也不再是 4 个字节，而是扩展为 8 个字节。“门”的意思是当异常或中断发生时，必须先通过相应门的检查，才可以进入异常或中断服务程序。

如图 6.5 所示为 IDTR 寄存器的结构。

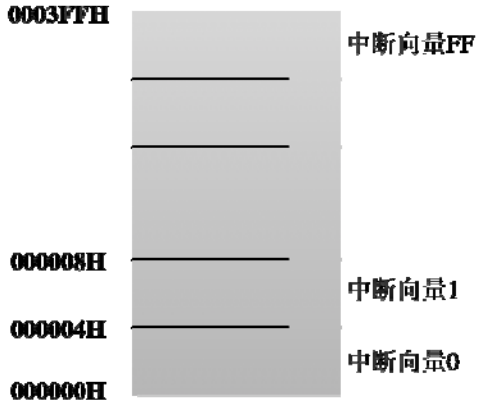

图 6.4　中断向量表

图 6.5　中断描述符表寄存器

IDTR 低 16 位保存 IDT 的大小，由此可见，保护模式下 IDT 最大可以有 64KB，即 8K 个表项。但由于 I386 只识别 256 个中断向量，所以 IDT 最大只能为 2KB，即包含 256 个表项，对应 256 个中断向量。

6.1.5　门

I386 在保护模式下 CPU 具有 4 种特权级（特权级 0、特权级 1、特权级 2 和特权级 3）。而 Linux 只使用了其中的两个，特权级 0 表示内核态，特权级 3 表示用户态，内核运行在内核态，用户应用程序运行在用户态。

异常与中断的发生会引起 CPU 运行状态的改变，比如发生除法错误异常时，CPU 就需要从特权级 3 切换到特权级 0。门的目的是在 CPU 希望进行状态切换时提供一种审查机制，保护系统运行。

门有 4 种，即任务门（Task Gate）、中断门（Interrupt Gate）、陷阱门（Trap Gate）和调用门（Call Gate），IDT 中包含了前 3 种门的描述符。

任务门的本意是用于任务切换的，在 Linux 中，任务即是进程，但描述一个进程所需的信息远远多于一个任务门所能表达的内容，所以 Linux 并不采用任务门来实现进程切换（门是一个硬件上的概念，事实上，2.0 版内核是使用任务门进行进程切换的）。实际上，内核里只有双重故障异常（中断向量 8）使用到了任务门。

中断门和陷阱门的描述符结构基本相同，如图 6.6 所示。

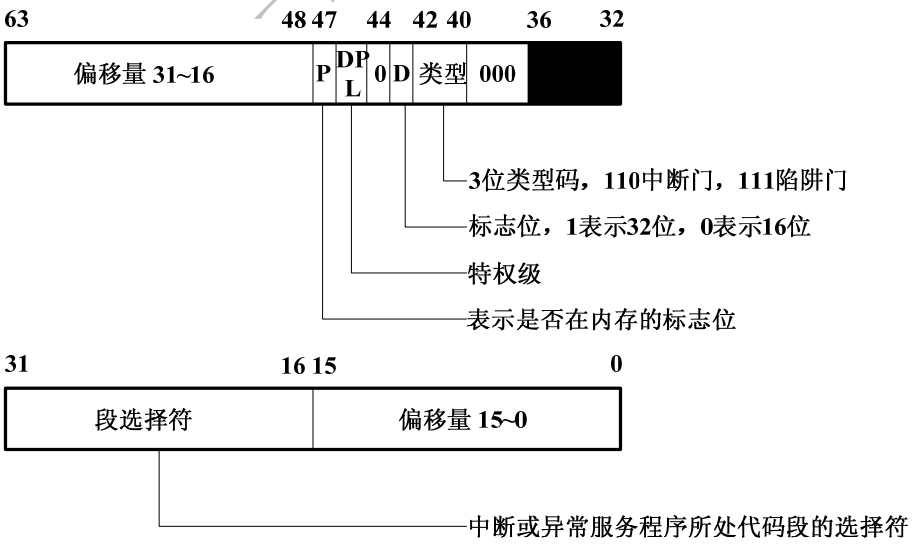

图 6.6　中断门与陷阱门的描述符结构

通过 16 位段选择符，可以从全局段描述符表（GDT）中获得异常或中断服务程序所处代码段的段描述符，再加上 32 位的段偏移量即可得到服务程序的地址。

华清远见

DPL（Descriptor Privilege Level）指定了门的特权级。门的一边是 CPU 的当前特权级（CPL），另一边是目标代码段的特权级，只有 CPL 小于等于门的 DPL，门才能够打开。通过门之后，还要将 CPL 与目标代码段的 DPL 相比较，只有 CPL 大于等于目标代码段的 DPL，才能够进入相应的服务程序。此时，目标代码段的 DPL 就变成 CPL，也就是说，通过门时，只能保持或提升 CPU 的特权级，而不能降低。

6.1.6 中断服务程序

中断服务程序必须在设备驱动程序中定义，并在使用 request_irq 函数申请 IRQ 线时，关联到所申请到的 IRQ 线上。它们都具有如下相同的原型。

```
/**
 * @irq:所申请到的 IRQ 线。
 * @dev_id:request_irq 函数传递进来的参数，具有全局唯一性，通常指向设备的私有
 * 数据结构，可以用其判断具体是哪个设备产生了中断。
 */
typedef irqreturn_t (*irq_handler_t)(int irq, void *dev_id);
```

中断服务程序的返回值是一个特殊类型——irqreturn_t，它在 include/linux/irqreturn.h 文件中定义。

```
typedef int irqreturn_t;

#define IRQ_NONE (0)
#define IRQ_HANDLED  (1)
#define IRQ_RETVAL(x) ((x) != 0)
```

irqreturn_t 只能取两个值：IRQ_NONE 表示不处理所收到的中断请求，比如当中断服务程序发现该中断请求并不由自己负责时；IRQ_HANDLED 表示接收到了有效的中断请求，并且做出了正确的处理。另外，也可以使用宏 IRQ_RETVAL(x)，若 x 为非 0 值，返回 IRQ_HANDLED，否则返回 IRQ_NONE。

中断服务程序是无需重入的。当一个给定的中断服务程序正在执行时，相应的中断线在所有处理器上都会被屏蔽，以避免在同一中断线上接收另一个新的中断。因此，同一个中断服务程序绝对不会被同时调用。这大大简化了中断服务程序的编写。

6.2 重要数据结构

内核中用于支持中断处理的数据结构主要有 3 个，分别是 irq_chip、irq_desc 和 irqaction，如图 6.7 所示为它们之间的关系。

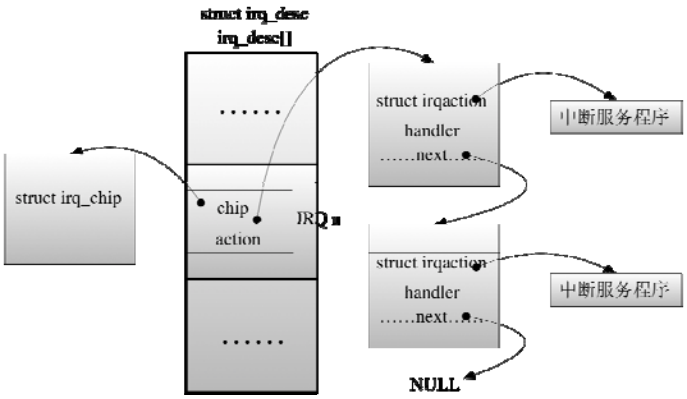

图 6.7 重要数据结构

6.2.1 中断描述符 irq_desc

数据结构 irq_desc 用于描述 IRQ 线的属性与状态，又被称为中断描述符（区别于之前所述的门描述符）。

每个 IRQ 都有它自己的 irq_desc 对象，所有的 irq_desc 对象组织在一起形成 irq_desc 数组，即中断描述符数组。

irq_desc 结构在 include/linux/irq.h 文件中定义如下。

代码清单 6.1　irq_desc 结构

```
struct irq_desc {
        /* 该 IRQ 线的公共服务程序 */
        irq_flow_handler_t   handle_irq;
        /* 该 IRQ 线所隶属的中断控制器 */
        struct irq_chip      *chip;
        struct msi_desc      *msi_desc;
        void         *handler_data;
        void         *chip_data;
        /* 描述该 IRQ 线所连接的特定中断源产生中断时，需要调用的服务程序。
           指向该 IRQ 线中断请求队列的头 */
        struct irqaction *action;
        /* 描述 IRQ 线状态的标志（见表 6.2） */
        unsigned int     status;

        /* 如果为 0，表示该 IRQ 线被激活；如果为一个正数，则表示该 IRQ 线被禁止的次数。
           通常称为禁止深度 */
        unsigned int     depth;
        unsigned int     wake_depth;
        /* 该 IRQ 线上发生中断的总次数 */
        unsigned int     irq_count;
        /* 该 IRQ 线上未处理中断发生的总次数 */
        unsigned int     irqs_unhandled;
        /* 上一次发生未处理中断时的 jiffies 值 */
        unsigned long        last_unhandled;
        /* irq_desc 对象都是全局的，因此需要使用锁进行保护 */
        spinlock_t       lock;
#ifdef CONFIG_SMP
        cpumask_t        affinity;
        unsigned int     cpu;
#endif
#if defined(CONFIG_GENERIC_PENDING_IRQ) || defined(CONFIG_IRQBALANCE)
        cpumask_t        pending_mask;
#endif
#ifdef CONFIG_PROC_FS
        struct proc_dir_entry    *dir;
#endif
        const char       *name;
} ____cacheline_internodealigned_in_smp;

   /* 声明一个中断描述符数组，共有 NR_IRQS（通常为 224）项，对应 NR_IRQS 个 IRQ 线 */
extern struct irq_desc irq_desc[NR_IRQS];
```

表 6.2　　IRQ 线状态标志

状态标志	描　　述
IRQ_INPROGRESS	正在处理，即该 IRQ 线的一个中断服务程序正在执行
IRQ_DISABLED	被禁止
IRQ_PENDING	已经被应答，但是由于某种原因，内核尚未对其进行处理
IRQ_REPLAY	IRQ 线已经被禁止，但前一个出现的中断还没有被应答
IRQ_AUTODETECT	该 IRQ 线被用来进行硬件设备探测
IRQ_WAITING	进行硬件设备探测时，会将所有没有挂载中断服务程序的 IRQ 线状态设置为 IRQ_WAITING，如果该 IRQ 上有中断产生，就清除这个状态，我们可以由此推断哪些中断引脚上产生过中断
IRQ_LEVEL	用于电平触发中断

IRQ_MASKED	被屏蔽
IRQ_PER_CPU	相应的中断服务程序只在同一 CPU 上运行
IRQ_NOPROBE	该 IRQ 线不用于硬件设备探测
IRQ_NOREQUEST	该 IRQ 线不能被申请
IRQ_NOAUTOEN	该 IRQ 线被申请成功后不会自动激活
IRQ_WAKEUP	该 IRQ 线上发生的中断能够唤醒系统
IRQ_MOVE_PENDING	中断需要被重新分发
IRQ_NO_BALANCING	执行中断负载均衡时，该 IRQ 线不被考虑在内

irq_count 与 irqs_unhandled 字段分别保存了中断发生的总次数与未处理中断的总次数。未处理中断是指没有被内核处理的中断，即与该 IRQ 线相关的所有中断处理程序都认为不是自己设备产生的中断，因此拒绝对其进行处理。

如果一个 IRQ 线上发生的未处理中断足够多，比如在发生第 100000 次中断时，未处理中断已经达到了 99900 次，则内核将禁止该 IRQ 线。

6.2.2 中断控制器描述符 irq_chip

数据结构 irq_chip 用于描述不同类型的中断控制器。Linux 可以支持多种中断控制器，所以内核分别创建了多个 irq_chip 结构对象，与之相对应，比如，I8259A 对应 i8259A_chip，APIC 对应 ioapic_chip、lapic_chip 等。

irq_chip 在 include/linux/irq.h 文件中的定义如下。

代码清单 6.2 irq_chip 结构

```
struct irq_chip {
        /* 中断控制器名称 */
        const char  *name;
        /* 启用与关闭指定的 IRQ 线 */
        unsigned int (*startup)(unsigned int irq);
        void    (*shutdown)(unsigned int irq);
        /* 激活与禁止指定的 IRQ 线 */
        void    (*enable)(unsigned int irq);
        void    (*disable)(unsigned int irq);

        /* 应答指定的 IRQ 线 */
        void    (*ack)(unsigned int irq);
        /* 屏蔽指定的 IRQ 线，阻塞它向 CPU 发送 */
        void    (*mask)(unsigned int irq);
        void    (*mask_ack)(unsigned int irq);
        /* 解除屏蔽指定的 IRQ 线 */
        void    (*unmask)(unsigned int irq);
        void    (*eoi)(unsigned int irq);

        /* 通知中断控制器，中断已经处理完毕 */
        void    (*end)(unsigned int irq);
        /* 绑定指定的 IRQ 到特定的 CPU 上 */
        void    (*set_affinity)(unsigned int irq, cpumask_t dest);
        /* 重新产生并发送指定的 IRQ */
        int     (*retrigger)(unsigned int irq);
        int     (*set_type)(unsigned int irq, unsigned int flow_type);
        /* 激活或者禁止 wake-on-interrupt 行为（该 IRQ 能否唤醒睡眠中的系统） */
```

```
    int     (*set_wake)(unsigned int irq, unsigned int on);

    /* Currently used only by UML, might disappear one day.*/
#ifdef CONFIG_IRQ_RELEASE_METHOD
    void    (*release)(unsigned int irq, void *dev_id);
#endif
    /*
     * For compatibility, ->typename is copied into ->name.
     * Will disappear.
     */
    const char  *typename;
};
```

6.2.3 中断服务程序描述符 irqaction

如前所述，多个设备可以共享同一个 IRQ 线，但是这些设备都有各自不同的中断处理方式。这就要求我们在进行中断处理时，能够区分共享同一个 IRQ 线的多个设备，基于这个目的，内核引入了数据结构 irqaction 来描述针对特定设备所产生中断的操作。

irqaction 在 include/linux/interrupt.h 文件中定义如下。

代码清单 6.3 irqaction 结构

```
typedef irqreturn_t (*irq_handler_t)(int, void *);

struct irqaction {
        /* 特定设备对应的中断服务程序 */
    irq_handler_t handler;
        /* 中断处理的标志（见表 6.3） */
    unsigned long flags;
    cpumask_t mask;
        /* 设备名*/
    const char *name;
        /* 设备的私有字段，通常用于标识设备，或指向设备驱动程序的数据，
           驱动程序申请 IRQ 时将其作为参数传递给中断处理程序 */
    void *dev_id;
        /* 指向 IRQ 线中断请求队列中的下一个 irqaction 对象 */
    struct irqaction *next;
        /* IRQ 线 */
    int irq;
    struct proc_dir_entry *dir;
};
```

表 6.3 主要的 irqaction 标志

状 态 标 志	描 述
IRQF_DISABLED	执行中断处理程序时，禁止对应的 IRQ 线
IRQF_SHARED	设备同意与其他设备共享 IRQ 线
IRQF_SAMPLE_RANDOM	可以被内核用作随机数产生器

共享同一个 IRQ 线的多个 irqaction 对象组成一个队列，即所谓的中断请求队列。中断产生时，在该 IRQ 线中断请求队列中挂载的所有中断服务程序将被依次调用执行。

6.3 中断子系统初始化

Linux 的中断处理机制主要包括以下 3 个方面的内容。

中断子系统初始化：内核在自身初始化过程中对中断处理机制的初始化，包括中断描述符表的初始化以及中断请求队列的初始化等内容。

中断或异常处理：一个实际中断或异常的处理过程。

中断 API：为设备驱动程序提供的一组 API，包括注册与释放、激活与禁止等函数。

本节首先介绍第一个方面的内容。

6.3.1 中断描述符表的初始化

中断描述符表 IDT 需要经过两个阶段的初始化，第一阶段初始化发生在内核引导过程，第二阶段初始化发生在内核初始化过程。

1．第一阶段初始化

第一阶段初始化主要完成两项工作：为 IDT 分配 2KB 大小的空间（256 个中断向量，每个中断向量对应一个 8bit 的门描述符），并初始化为默认值；存储 IDT 的起始地址到 IDTR 寄存器。

相关代码位于 arch/i386/kernel/head.S 文件：

```
268 call setup_idt /* 初始化 IDT */
……
334 lidt idt_descr /* 存储 IDT 的起始地址到 IDTR 寄存器中 */
```

第 268 行调用汇编函数 setup_idt 完成 IDT 的初始化：

```
393 setup_idt:
        /* 将默认处理程序 ignore_int 函数的地址存储在 edx 寄存器 */
394      lea ignore_int,%edx
        /* 将代码段的段选择符存储在 eax 寄存器的高 16 位 */
395      movl $(__KERNEL_CS << 16),%eax
        /* 将 edx 寄存器低 16 位存储在 eax 寄存器低 16 位 */
396      movw %dx,%ax
        /* 将值 0x8E00 作为门描述符的默认属性存储在 edx 寄存器低 16 位 */
397      movw $0x8E00,%dx
398
        /* 将 IDT 的起始地址（即第一个门描述符地址）存储在 edi 寄存器 */
399      lea idt_table,%edi
        /* 将中断向量的数目 256 存储在 ecx 寄存器 */
400      mov $256,%ecx
```

上面的代码执行之后，edx 与 eax 寄存器的分布如图 6.8 所示：

图 6.8　edx 与 eax 寄存器的分布

然后，即可使用 edx 与 eax 寄存器作为一个门描述符的高 32 位与低 32 位，初始化 IDT 的每一项。

```
401 rp_sidt:
        /* 存储 eax 寄存器的值到门描述符的低 32 位（edi 寄存器存储了 IDT 的地址） */
```

```
402     movl %eax,(%edi)
       /* 存储 edx 寄存器的值到门描述符的高 32 位 */
403     movl %edx,4(%edi)
       /* 使 edi 寄存器指向下一个门描述符 */
404     addl $8,%edi
       /* ecx 寄存器值减 1，循环执行上面的步骤，直到 IDT 的 256 个门描述符初始化完毕 */
405     dec %ecx
406     jne rp_sidt
```

默认处理程序 ignore_int 并不做什么实际的工作，也从不应该被调用，如果因为某个硬件或软件问题而被调用到，则会打印出“Unknown interrupt or fault at EIP …”。

2．第二阶段初始化

内核在自身初始化过程的 start_kernel 函数中，使用了两个函数 trap_init 和 init_IRQ 来完成 IDT 的第二阶段初始化。其中 trap_init 函数完成对系统保留中断向量（异常、非屏蔽中断以及系统调用）的初始化，init_IRQ 函数则完成其余中断向量的初始化。

（1）函数 trap_init。

函数 trap_init 在 arch/i386/kernel/trap.c 文件中的定义如下。

代码清单 6.4　对系统保留中断向量的初始化

```
void __init trap_init(void)
{
    ……
      /* 初始化中断向量 0~19，包括异常和不可屏蔽中断 */
  set_trap_gate(0,&divide_error);
  set_intr_gate(1,&debug);
  set_intr_gate(2,&nmi); /* 不可屏蔽中断 */
  set_system_intr_gate(3, &int3); /* int 3/4 能被所有进程调用 */
  set_system_gate(4,&overflow);
  set_trap_gate(5,&bounds);
  set_trap_gate(6,&invalid_op);
  set_trap_gate(7,&device_not_available);
  set_task_gate(8,GDT_ENTRY_DOUBLEFAULT_TSS);
  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);
#ifdef CONFIG_X86_MCE
  set_trap_gate(18,&machine_check);
#endif
  set_trap_gate(19,&simd_coprocessor_error);
    ……
      /* 初始化中断向量 0x80，即系统调用 */
  set_system_gate(SYSCALL_VECTOR,&system_call);
    ……
```

```
}
```

通过 trap_init 函数的定义可以看出，内核主要使用了 set_intr_gate、set_trap_gate、set_system_gate、set_system_intr_gate 以及 set_task_gate 等 5 个函数初始化 IDT 中的门描述符。

代码清单 6.5　门描述符初始化函数

```
/* 设置中断门描述符，n 为中断向量号，addr 为服务程序地址 */
void set_intr_gate(unsigned int n, void *addr)
{
    _set_gate(n, DESCTYPE_INT, addr, __KERNEL_CS);
}
/* 设置特权级为 3 的中断门描述符（中断门描述符特权级默认为 0） */
static inline void set_system_intr_gate(unsigned int n, void *addr)
{
_set_gate(n, DESCTYPE_INT | DESCTYPE_DPL3, addr, __KERNEL_CS);
}
/* 设置陷阱门描述符 */
static void __init set_trap_gate(unsigned int n, void *addr)
{
    _set_gate(n, DESCTYPE_TRAP, addr, __KERNEL_CS);
}
/* 设置系统门描述符 */
static void __init set_system_gate(unsigned int n, void *addr)
{
    _set_gate(n, DESCTYPE_TRAP | DESCTYPE_DPL3, addr, __KERNEL_CS);
}
/* 设置任务门描述符 */
static void __init set_task_gate(unsigned int n, unsigned int gdt_entry)
{
_set_gate(n, DESCTYPE_TASK, (void *)0, (gdt_entry<<3));
}
```

set_intr_gate、set_trap_gate、set_system_gate、set_system_intr_gate 以及 set_task_gate 函数都通过调用 _set_gate 函数设置门描述符，区别只是依据门类型的不同，而使用不同的参数。

_set_gate 函数在 include/asm-i386/desc.h 文件中定义如下。

```
/*
 * 设置门描述符的底层函数，gate 为中断向量，type 为门描述符的属性，
 * addr 为处理程序地址，seg 为段选择符。
 */
static inline void _set_gate(int gate, unsigned int type, void *addr, unsigned short seg)
{
    __u32 a, b;
   /* pack_gate 函数将参数封装成一个门描述符保存在变量 a、b 中。a 保存低 32 位，
    * 其中低 16 位为 addr 的低 16 位，高 16 位为段选择符 seg。b 保存高 32 位，其中低 16 位
    * 为 type，高 16 位为 addr 的高 16 位。
    */
    pack_gate(&a, &b, (unsigned long)addr, seg, type, 0);
   /* 将 pack_gate 获得的门描述符写入 IDT，idt_table 为第一阶段初始化中预留
    * 的 IDT 地址
    */
    write_idt_entry(idt_table, gate, a, b);
}
```

Linux 本身并不使用调用门(因为一些 UNIX 变种使用调用门来实现系统调用，为了使应用程序在 Linux 和这些 UNIX 变种上可兼容，Linux 相应也在局部描述符 LDT 中设置了几个调用门)，对于任务门，也只有双重故障异常（中断向量 8）使用到。

Linux 主要使用的是中断门和陷阱门，陷阱门又被细分为陷阱门和系统门。

中断门使用 set_intr_gate 函数设置，传递给_set_gate 函数的 type 参数值为 DESCTYPE_INT。

DESCTYPE_INT 表示中断门描述符，特权级为 0。因此，对于用户应用程序来说，因为 CPL 为 3，大于中断门的 DPL，所以不可能通过某种途径，比如调用一些汇编指令，穿过中断门去进入中断服务程序。而对于中断来说，中断门的 DPL 被忽略，总是能通过。

陷阱门使用 set_trap_gate 函数设置，传递给_set_gate 函数的 type 参数值为 DESCTYPE_TRAP。DESCTYPE_TRAP 表示陷阱门，特权级也为 0。

系统门使用 set_system_gate 函数设置，与陷阱门不同的是特权级为 3，因此用户应用可以通过调用某些指令，穿过系统门进入中断服务程序。

最为常见的系统门是系统调用（中断向量 0x80），用户应用通过执行“int 0x80”指令陷入内核，穿过系统门执行系统调用服务例程。

（2）函数 init_IRQ。

函数 init_IRQ 在 arch/i386/kernel/paravirt.c 文件中定义如下。

```
void init_IRQ(void)
{
    paravirt_ops.init_IRQ();
}
```

paravirt-ops 接口规范由 IBM、VMware、Red Hat 和 XenSource 组成的联合小组共同开发，它包含了很多 VMI（Virtual Machine Interface，虚拟机接口）里面提出的概念，用于支持透明半虚拟化。从 2.6.20 开始，paravirt-ops 已经成为官方 Linux 内核的一部分。

paravirt_ops 的 init_IRQ 成员被初始化 native_init_IRQ 函数，在 arch/i386/kernel/I8259.c 文件中定义如下。

代码清单 6.6　其余中断向量的初始化

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

    /* all the set up before the call gates are initialised */
    /* 初始化中断控制器以及初始化每个 IRQ 线的中断请求队列。 */
    pre_intr_init_hook();

    /*
     * Cover the whole vector space, no vector can escape
     * us. (some of these will be overridden and become
     * 'special' SMP interrupts)
     */
    /* NR_VECTORS 为 256，FIRST_EXTERNAL_VECTOR 为 32（表示预留的中断向量 0～31），
     * NR_IRQS 为 224，因此此处的 for 循环对其余 224 个中断向量进行初始化。
     */
    for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
        int vector = FIRST_EXTERNAL_VECTOR + i;
        if (i >= NR_IRQS)
            break;
        /* 跳过用于系统调用的中断向量，因为在 trap_init 函数中已经初始化。*/
        if (vector != SYSCALL_VECTOR)
            set_intr_gate(vector, interrupt[i]);
```

```
405     }
406
407     /* setup after call gates are initialised (usually add in
408      * the architecture specific gates)
409      */
        /* 主要针对 SMP 系统进行一些额外的初始化。 */
410     intr_init_hook();
411
412     /*
413      * External FPU? Set up irq13 if so, for
414      * original braindamaged IBM FERR coupling.
415      */
416     if (boot_cpu_data.hard_math && !cpu_has_fpu)
417         setup_irq(FPU_IRQ, &fpu_irq);
418
        /* 如果配置内核时，选上了 CONFIG_4KSTACKS 选项，则为每个 CPU 分配一个 4kB 大小的栈，
         * 专门用于中断和异常处理。否则，中断与异常处理时与被中断进程共用一个 8kB 大小的内核栈。
         */
419     irq_ctx_init(smp_processor_id());
420 }
```

第 404 行将位于 32～255 之间，除去系统调用外的其他中断向量的中断处理程序设置为 interrupt[i]。interrupt 数组共有 NR_IRQS 项，在 arch/i386/kernel/entry.S 文件中定义并初始化：

```
582 .data
583 ENTRY(interrupt)
584 .text
585
586 ENTRY(irq_entries_start)
587     RING0_INT_FRAME
588 vector=0
    /* 循环 NR_IRQS 次，编译时将展开 */
589 .rept NR_IRQS
590     ALIGN
591  .if vector
592     CFI_ADJUST_CFA_OFFSET -4
593  .endif
594 1:  pushl $~(vector)
595     CFI_ADJUST_CFA_OFFSET 4
596     jmp common_interrupt
597  .previous
        /* 存储着指向 594 行的地址，随着循环的展开，每次的值都不同 */
598     .long 1b
599  .text
600 vector=vector+1
601 .endr
602 END(irq_entries_start)
603
604 .previous
605 END(interrupt)
606 .previous
```

第 583 行和 598 行都属于数据段，虽然从代码表面上看它们是分隔开的，但实际上却被编译器安排在了连续的数据段内存中。同时由于第 589 行与 601 行之间的代码要循环执行 NR_IRQS 次，因此 598 行也将被执行 NR_IRQS 次，代码展开后，数据段的内容就近似于如下的形式。

```
ENTRY(interrupt)
    .long 1b/* 0 */
    .long 1b
    ……
    .long 1b     /* NR_IRQS */
```

即构成一个共有 NR_IRQS 项的 interrupt 数组。

对于代码段，位于第 586～601 行的指令序列也是连续存储的，第 589 行的循环执行 NR_IRQS 次后，共形成 NR_IRQS 个代码片断。因为第 598 行存储的是第 594 行代码的地址，所以 interrupt 数组的每一项实际是指向了其中的一个代码片段，形式如下：

```
pushl $~(vector)
CFI_ADJUST_CFA_OFFSET 4
jmp common_interrupt
```

它们最终会统一调用一个公共的中断处理函数 common_interrupt。

对于中断向量 32，即 IRQ0，对应的中断处理程序 interrupt[0]即是：

```
pushl $~(0)
CFI_ADJUST_CFA_OFFSET 4
jmp common_interrupt
```

6.3.2 中断请求队列的初始化

init_IRQ 函数通过调用 pre_intr_init_hook 函数，并最终调用 init_ISA_irqs 函数初始化中断控制器以及每个 IRQ 线的中断请求队列。

init_ISA_irqs 在 arch/i386/kernel/i8259.c 文件中定义如下。

代码清单 6.7 中断控制器 8259A 初始化

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

/* 初始化 APIC 或 8259A 中断控制器 */
#ifdef CONFIG_X86_LOCAL_APIC
    init_bsp_APIC();
#endif
    init_8259A(0);

    /* 初始化中断描述符数组。*/
    for (i = 0; i < NR_IRQS; i++) {
        irq_desc[i].status = IRQ_DISABLED;
        irq_desc[i].action = NULL;
        irq_desc[i].depth = 1;

        /* 将 IRQ0～IRQ15 的中断控制器初始化为 8259A，其余的初始化为 no_irq_chip。*/
        if (i < 16) {
            /*
             * 16 old-style INTA-cycle interrupts:
             */
            set_irq_chip_and_handler_name(i, &i8259A_chip,
                                          handle_level_irq, "XT");
```

```
375         } else {
376             /*
377              * 'high' PCI IRQs filled in on demand
378              */
379             irq_desc[i].chip = &no_irq_chip;
380         }
381     }
382 }
```

每个 IRQ 的中断请求队列在第 366 行被初始化为空，因此，此时即使真的有中断发生，也并没有实际的中断服务程序为其服务。

为了使中断得到真正的处理，需要在设备驱动程序中使用 request_irq 函数注册一个 IRQ。request_irq 函数会为设备创建 irqaction 结构对象，并将其添加到该 IRQ 的中断请求队列中，之后，设备产生中断时，就会通过该 IRQ 的中断请求队列找到已经添加的中断服务程序进行处理。

6.4 中断或异常处理

如图 6.9 所示为中断或异常处理的基本过程。

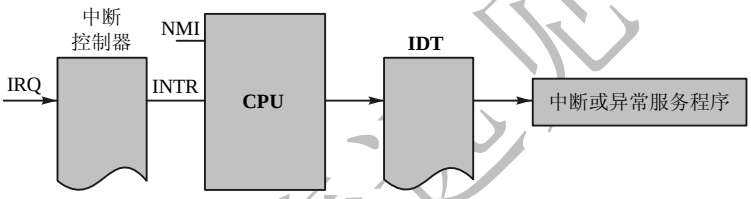

图 6.9　中断和异常处理基本过程

中断处理基本过程：首先设备产生中断，并通过中断线将中断信号送往中断控制器。如果该中断没有被屏蔽，则会被送往 CPU 的 INTR 引脚。CPU 立即停止当前的工作，根据从中断控制器获得的中断向量号，从 IDT 中找到相应的门描述符，从而获取中断服务程序的地址并执行。

异常处理基本过程：与中断不同，异常并不需要经过中断控制器转发电信号。当异常发生时，CPU 通过其特定的中断向量号，从 IDT 中查找相应的门描述符，并获取异常服务程序的地址。

6.4.1　中断控制器的工作

中断控制器是外部设备和 CPU 之间中断信号传递的桥梁，如图 6.10 所示为中断控制器 8259A 在中断处理过程中所做的工作。

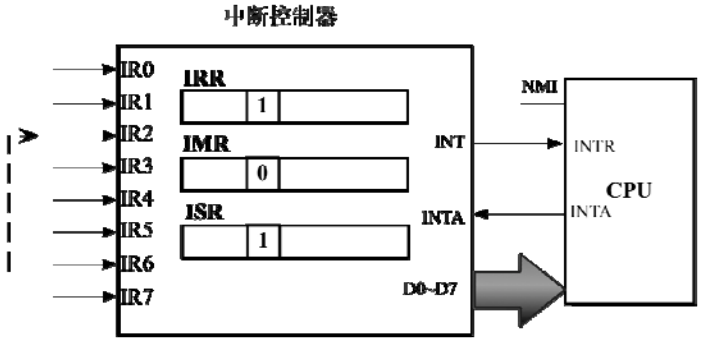

图 6.10　8259A 对中断的处理

8259A 主要有 3 个寄存器用于对中断的响应与处理。

IRR：即 Interrupt Request Register，中断请求寄存器，共 8bit，对应 IR0～IR7 八个中断引脚。当某个引脚的中断请求到来后，若该引脚没有被屏蔽，则 IRR 中对应的 bit 被置为 1，表示已经收到设备的中断请求，但还未提交给 CPU。

ISR：即 Interrupt Service Register，中断服务寄存器，共 8bit，对应 IR0～IR7 八个中断引脚。当 IRR 中的某个中断请求被送往 CPU 后，ISR 中对应的 bit 被置为 1，表示中断已提交给 CPU，但 CPU 还未处理完。

IMR：即 Interrupt Mask Register，中断屏蔽寄存器，共 8bit，对应 IR0～IR7 八个中断引脚。当某个 bit 置为 1 时，对应的中断引脚被屏蔽。

8259A 的各个中断引脚之间有一定的优先级，同时也有两种优先级管理方式：固定优先级与循环优先级。固定优先级指优先级固定不变，号码越小优先级越高。循环优先级指优先级在系统工作过程中可以动态改变。

8259A 支持两种中断嵌套模式：一般嵌套模式与特殊嵌套模式。一般嵌套模式是指，当一个中断正在被 CPU 处理时，优先级等于或低于该中断的中断被自动屏蔽，而更高优先级的中断则会被立即送往 CPU。特殊嵌套模式与一般嵌套模式的区别为，当一个中断正在被 CPU 处理时，同级的中断不被屏蔽。Linux 默认使用一般嵌套模式。

下面是 8259A 在中断处理过程中所要完成的工作。

中断请求。设备发起中断时，与其相连的 IR 引脚上产生电信号，若对应的中断没有被屏蔽，即 IMR 中相应位为 0，则设置 IRR 中的相应位为 1，并通过 INT 引脚向 CPU 的 INTR 引脚发出中断请求信号。若已经被屏蔽，则仅仅丢弃该中断请求。

中断响应。CPU 响应中断必须满足 3 个条件：至少有一个中断请求；CPU 允许中断；当前指令执行完毕。因此，当 8259A 向 CPU 提交中断请求信号时，CPU 可能正在执行一条指令，并不会立即进行响应。此时，可能有其他 IRQ 线也产生了中断请求，如果没有被屏蔽，那么这些请求对应的 IRR 位也会被设置为 1。

而当 CPU 执行完一条指令时，会去检查 INTR 管脚是否有信号，即是否有新的中断请求，如果有，还要检查 EFLAGS 寄存器的中断允许标志 IF 是否为 1，若 IF 为 1，则通过 INTA 引脚应答 8259A，表示收到中断请求。

优先级判定。8259A 收到 CPU 的应答后，在 IRR 中挑选优先级最高的中断，将其在 ISR 中的对应位置 1，并将 IRR 中相应位置 0，表明此中断正在接受 CPU 的处理。

提交中断向量。CPU 通过 INTA 引脚第二次发出脉冲，8259A 收到后根据被设置的起始向量号（通过初始化命令寄存器 ICW 被初始化），计算最高优先级中断的中断向量（比如起始向量号为 16，当前中断请求为 IR3，则得到的中断向量为 16+3=19），并将它通过数据线 D0～D7 提交给 CPU。

中断结束。提交中断向量之后，8259A 会检测是否 AEOI（Auto EOI）模式，若是，则自动清除中断请求信号，将 ISR 中的相应位清零。若不是，则需要等待 CPU 发送 EOI（End of Interrupt，CPU 通知 PIC 中断处理结束的方式，由中断服务程序发起）指令。收到 EOI 后，ISR 中的相应位才会被清零。

上述过程中，需要注意的是，如果一个中断请求被屏蔽，则它将会被丢弃，从而失去参加优先级判定的机会。

另外需要注意的是，对中断请求的优先级判定位于 8259A 中。假如 CPU 正在处理 IRQ1 线来的中断请求，此时 ISR 中 IRQ1 的相应位为 1。如果这时有一个来自 IRQ2 的中断请求产生，8259A 会将其同 ISR 中的值进行比较，发现比它优先级高的 IRQ1 所对应的位已经被置位，因此仅设置 IRR 中的相应位。如果这时来的是 IRQ0，与 ISR 比较后，8259A 会立即将 ISR 中 IRQ0 对应的位置 1，并向 CPU 提交中断请求。因为 CPU 还正在处理 IRQ1，所以这时 ISR 中 ISR0 与 IRQ1 的相应位都为 1。

如果 8259A 工作于 AEOI 模式，那么 ISR 中的位总是被清零的（在 AEOI 模式下，8259A 只要向 CPU 提交了中断向量就会自动将 ISR 中的相应位清零），这就意味着如果有新的中断请求产生，8259A 就会立即向 CPU 提交中断请求，即使 CPU 正在处理 IRQ0 的中断，CPU 只会简单地应答 8259A。因此，这种情况下低优先级的中断就可能会中断高优先级的中断服务程序。

华清远见

6.4.2 CPU 的工作

如图 6.11 所示为中断或异常发生时，CPU 所要做的工作。

确定中断或异常的中断向量 *i*（在 0～255 之间）。可屏蔽中断的中断向量从中断控制器获得，不可屏蔽中断和异常的中断向量是固定的。

通过 IDTR 寄存器找到 IDT，读取第 *i* 项（即第 *i* 个门描述符）。

进行特权级检查。将 CPU 的当前特权级 CPL 与门描述符的 DPL 相比较，如果 CPL 小于 DPL，则能够通过门，否则产生通用保护异常（中断向量 13）。通过门之后，获取门描述符中段选择符所指向代码段的 DPL，将其与 CPL 相比较，如果小于，则执行中断处理程序，否则产生一个通用保护异常。

若上述过程检测到特权级发生了变化，则需要进行堆栈的切换。因为中断或异常服务程序运行在内核态（特权级为 0），当中断或异常在 CPU 处于用户态（特权级为 3）时发生，将引起堆栈的切换。也就是说，从用户态堆栈切换到内核态堆栈。

如果当前是异常，则 CPU 将异常码压入当前的堆栈。

如果当前是中断，CPU 清零 EFLAGS 的 IF 位，即关闭所有的可屏蔽中断；如果是异常，CPU 不会清零该位。

进入中断或异服务程序并执行。

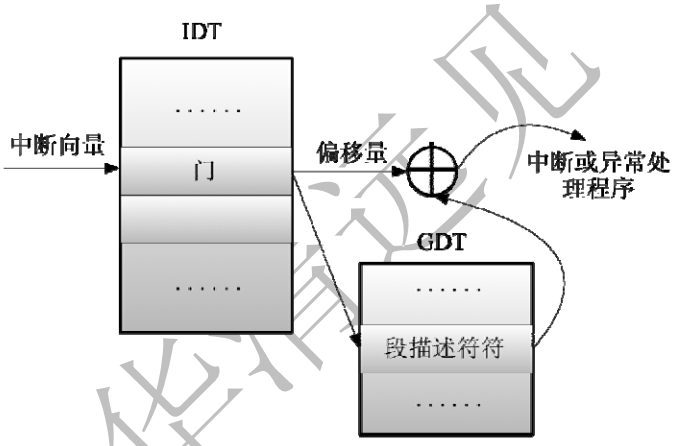

图 6.11　CPU 对中断或异常的处理

6.4.3 内核对中断的处理

图 6.12 所示为中断在内核中的处理过程。

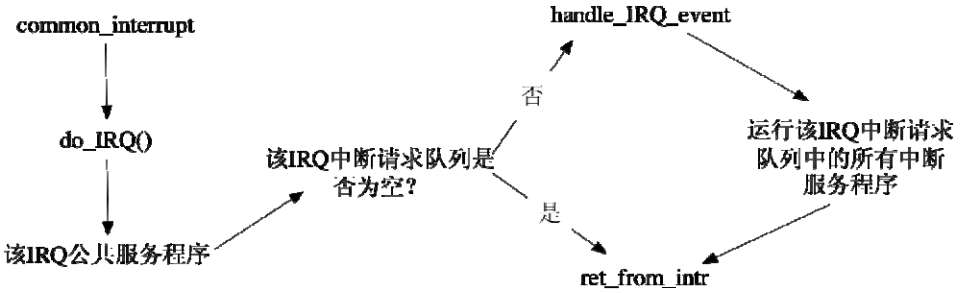

图 6.12　内核对中断的处理

所有中断（除去不可屏蔽中断）的服务程序在 init_IRQ 函数中都被初始化为 interrupt[i]，interrupt 数组的每一项指向一个代码片段，该代码片段除了将中断向量号压入堆栈外，还要调用到一个公共的处理程序 common_interrupt。

```
common_interrupt:
    SAVE_ALL
```

```
615     TRACE_IRQS_OFF
616     movl %esp,%eax
617     call do_IRQ
618     jmp ret_from_intr
```

common_interrupt 函数首先保存现场，将中断发生前所有寄存器的值（属于被中断的进程）保存在堆栈中，然后调用 do_IRQ 函数。

do_IRQ 函数在 arch/i386/kernel/irq.c 文件中定义为：

```
071 fastcall unsigned int do_IRQ(struct pt_regs *regs)
072 {
073     ……
        /* 取得中断向量号 */
075     int irq = ~regs->orig_eax;
076     ……
        /* 进入中断上下文 */
089     irq_enter();
090     ……
        /* 调用该 IRQ 的公共处理程序对中断进行处理 */
143     desc->handle_irq(irq, desc);
144
        /* 退出中断上下文 */
145     irq_exit();
147     ……
148 }
```

从堆栈中取出中断向量号之后，do_IRQ 函数调用 irq_enter 进入中断上下文。irq_enter 函数主要的工作是增加被中断进程的 preempt_count 计数器的值，表示禁止抢占。在中断处理结束调用 irq_exit 退出中断上下文时，会相应地减少该计数器的值。

do_IRQ 函数的工作重心在第 143 行，调用每个 IRQ 线的公共服务程序。

在通用 IRQ 层引入之前，每个 IRQ 线并没有自己单独的服务程序，而是所有的 IRQ 线都共用一个公共的服务程序__do_IRQ 对中断进行处理。

在通用 IRQ 层引入之后，对不同类型中断（边沿触发、电平触发等）的处理流程做出了区分，并分别设置了一些对应的服务程序，比如边沿触发中断对应 handle_edge_irq 函数，电平触发中断对应 handle_level_irq 函数等。

无论是旧的__do_IRQ 函数还是新的各种类型中断单独的服务程序，它们的基本原理还是一样的。都要首先屏蔽当前的 IRQ，禁止该中断线上的中断传递。

然后，会判断该 IRQ 线的中断请求队列上是否为空，即是否有一个或多个有效的中断服务程序，如果有的话，就调用 handle_IRQ_event 函数来遍历并执行挂载在该 IRQ 线中断请求队列上的所有中断处理程序。

handle_IRQ_event 函数在 arch/kernel/irq/handle.c 文件中定义为：

```
irqreturn_t handle_IRQ_event(unsigned int irq, struct irqaction *action)
{
    irqreturn_t ret, retval = IRQ_NONE;
    unsigned int status = 0;

    handle_dynamic_tick(action);

    /* 因为 CPU 在通过中断门时自动将中断关闭，如果这里希望将它们打开，即希望
     * 中断服务程序能够在中断打开的情况下执行，就需要在使用 request_irq 函数
     * 注册中断处理程序时，不要设置 IRQF_DISABLED 标志。
     */
    if (!(action->flags & IRQF_DISABLED))
        local_irq_enable_in_hardirq();
```

```
138
        /* 遍历该 IRQ 线的中断请求队列，并调用其中的所有中断服务程序。通常每个具体的
         * 中断服务程序中都会一开始检查各自的中断源，判断是否有来自该设备的中断请求，
         * 如果没有则马上退出。而且每个 IRQ 线的中断请求队列一般也不会很大，所以这个
         * 过程并不会耗费很长的时间。
         */
139     do {
140         ret = action->handler(irq, action->dev_id);
141         if (ret == IRQ_HANDLED)
142             status |= action->flags;
143         retval |= ret;
144         action = action->next;
145     } while (action);
146
        /* 如果注册 IRQ 时设置了 IRQF_SAMPLE_RANDOM 标志，则需要调用
         * add_interrupt_randomness 函数，以中断间隔时间为随机数产生熵。
         */
147     if (status & IRQF_SAMPLE_RANDOM)
148         add_interrupt_randomness(irq);
        /* 最后关闭中断，函数返回。*/
149     local_irq_disable();
150
151     return retval;
152 }
```

中断处理结束之后，do_IRQ 函数返回到 common_interrupt，然后调用汇编函数 ret_from_intr 从中断返回。

如果内核正在从中断返回到用户空间，并且被中断进程设置了标志 TIF_NEED_RESCHED，则 ret_from_intr 会调用 schedule 函数进行重新调度。如果内核正在返回内核空间，即中断了内核本身，则 ret_from_intr 会检查被中断进程的 preempt_count 计数器是否为 0，只有为 0 时，schedule 函数才会被调用。

6.4.4 内核对异常的处理

与中断不同，各种异常都具有固定的中断向量以及固定的异常服务程序（在 trap_init 函数中指定）。因此，当异常发生时，将直接跳转到相应的服务程序并执行。

异常服务程序都定义在 arch/i386/kernel/entry.S 文件，比如堆栈段异常（中断向量 12）的服务程序即是：

```
955 ENTRY(stack_segment)
956     RING0_EC_FRAME
957     pushl $do_stack_segment
958     CFI_ADJUST_CFA_OFFSET 4
959     jmp error_code
960     CFI_ENDPROC
961 END(stack_segment)
```

中断发生时，会跳转到公共的服务程序 common_interrupt。异常也有类似的操作，只不过是跳转到汇编函数 error_code。

error_code 函数会调用真正的服务程序对异常进行处理。对于堆栈段异常，这个服务程序就是 957 行的 C 函数 do_stack_segment。

与中断有所不同的是，如果所发生的异常产生出错码，就将这个出错码也压入堆栈。而对于不产生出错码的异常，则在进入 error_code 函数前补上一个。比如无效操作码异常（中断向量 6）：

```
919 ENTRY(invalid_op)
920     RING0_INT_FRAME
```

```
921     pushl $0
922     CFI_ADJUST_CFA_OFFSET 4
923     pushl $do_invalid_op
924     CFI_ADJUST_CFA_OFFSET 4
925     jmp error_code
926     CFI_ENDPROC
927 END(invalid_op)
```

第 921 行的 0，即是出错码的替代品。

对异常处理完毕后，error_code 会调用汇编函数 ret_from_exception 从异常返回。

6.5 中断 API

内核提供了一组 API 接口用于控制系统上的中断状态，了解并掌握这些接口的使用是我们进行驱动开发的基本功。

6.5.1 注册和释放

IRQ 线是一个非常宝贵的资源，驱动程序在使用前必须要先注册申请，不再使用时必须要释放申请的 IRQ。

request_irq 函数和 free_irq 函数分别用来注册和释放 IRQ，它们在 include/linux/interrupt.h 文件中声明如下。

```
/**
 * @irq: 要申请的 IRQ 号。很多设备使用的 IRQ 号通常都是预先分配好的，比如系统时钟和键盘。
 * @handler: 要注册的中断服务程序。
 * @irqflags: 中断的类型标志。它可以取下面三个值之一，或者它们的“|”：
 *   IRQF_SHARED: 表示可以与其他设备共享同一条中断线。
 *   IRQF_DISABLED: 在本地 CPU 上，中断处理程序在禁止所有中断的情况下执行，
 *        可以不受其他中断干扰。如果没有设置这个标志，则只有该中断处理程序对应的
 *        那条中断线被屏蔽，而其他中断都是激活的。
 *   IRQF_SAMPLE_RANDOM: 表明这个中断能够用来产生内核熵。
 * @dev_name: 中断源的名称，可以在/proc/interrupts 文件中看到。
 * @dev_id: 传递给中断服务程序 handler 的参数，必须全局唯一，通常都是用来指向设备
 * 的私有数据结构，以便标识具体是哪个设备产生了中断。如果不与其他设备共享中断线，
 * 则可以指定为 NULL。
 */
int request_irq(unsigned int irq, irq_handler_t handler,
        unsigned long irqflags, const char *devname, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

1. request_irq 函数

request_irq 的主要任务是为 IRQ 线的中断请求队列创建 irqaction 节点。因为中断请求队列在中断子系统的初始化过程中被初始化为空，所以如果设备没有使用 request_irq 函数为其填充节点，即使设备产生了中断，也得不到任何真正的处理。

参数 irqflags 在旧的内核版本中，使用的是 SA_SHIRQ、SA_INTERRUPT 和 SA_SAMPLE_RANDOM，但是在 2007 年 9 月，它们分别被 IRQF_SHARED、IRQF_DISABLED 和 IRQF_SAMPLE_RANDOM 所取代。为了保持兼容，旧的标志仍然能够使用，但新的代码建议使用新的标志。

request_irq 返回 0 表示成功，此时该 IRQ 被激活。返回非 0 值，表示发生了错误，常见的错误为-EBUSY，表示指定的 IRQ 已经被占用，或者是没有设置 IRQF_SHARED 标志（与其他设备共享同一条中断线时）。

内核接收一个中断后，将依次调用在该中断线上注册的每一个中断服务程序。设备驱动程序必须知道它是否应该为这个中断请求负责。因此，一个中断服务程序如果判断与它相关的设备并没有产生该中断请求，那么它应该立即退出。通常硬件设备都会提供状态寄存器(或类似机制)，以供中断服务程序进行检查。

为了避免中断处理程序在设备初始化完成之前就开始执行，初始化硬件和调用 request_irq 注册 IRQ 线的顺序必须正确。

2．free_irq 函数

卸载设备驱动程序时，必须调用 free_irq 函数删除中断请求队列中的相应 irqaction 节点，并释放中断线。

如果 IRQ 线不是共享的，则 free_irq 删除 irqaction 节点之后，禁止该 IRQ 线。否则，free_irq 只删除参数 dev_id 指定的 irqaction 节点，而该 IRQ 线在中断请求队列中的所有 irqaction 节点都已经被删除时，才会被禁止。

free_irq 直到该 IRQ 线上的所有中断服务程序都执行完毕时才返回。

6.5.2 激活和禁止

我们通常需要通过禁止中断线来确保某个中断服务程序不会抢占当前的代码，以免数据的并发访问产生问题。但是对于 MP 系统，我们只能防止来自当前 CPU 的其他中断服务程序的并发访问，对于来自其他 CPU 的则需要通过锁机制进行保护。

如表 6.4 所示，内核提供了一组函数用于操作中断的状态，比如禁止当前 CPU 上的中断处理，或者仅仅禁止某个指定的中断线。

表 6.4 激活和禁止 IRQ 的函数

API 函数	描　述
local_irq_disable ()	禁止当前 CPU 的中断
local_irq_enable()	激活当前 CPU 的中断
local_irq_save(unsigned long)	禁止当前 CPU 的中断并保存标志寄存器的内容
local_irq_restore(unsigned long)	恢复当前处理器的中断状态，必须与 local_irq_save 在同一个函数中被调用
disable(unsigned int irq)	禁止指定的中断线，在函数返回前，确保该中断线上没有中断服务程序在执行
enable_irq(unsigned int irq)	激活指定的中断线
disable_irq_nosync (unsigned int irq)	调用 irq_chip 的 disable 成员禁止指定的中断线，但并不确保该中断线上所有已开始执行的中断服务程序已全部退出

对于 X86 来说，local_irq_disable 函数仅仅是通过 cli 指令将当前 CPU 标志寄存器中的中断标志位 IF 清 0，使该 CPU 不响应来自 INTR 引脚的中断请求。而 local_irq_enable 函数则是通过 sti 指令将当前 CPU 标志寄存器中的中断标志位 IF 置 1，允许该 CPU 响应来自 INTR 引脚的中断请求。

disable、enable_irq 与 disable_irq_nosync 可以嵌套调用，但必须注意，每次调用 disable 和 disable_irq_nosync，都必须相应调用一次 enable_irq。

通常情况下禁止多个设备共享的中断线是不合适的，所以我们一般都不需要使用 disable 和 disable_irq_nosync。

6.5.3 其他 API 函数

华清远见

如表 6.5 所示为内核提供的其他一些 API 函数。

表 6.5　　其他 API 函数

API 函数	描　　述
in_irq()	判断内核是否正在执行中断服务程序
in_softirq()	判断内核是否正在处理软中断
in_interrupt()	判断内核是否处于中断上下文，如果内核正在执行中断服务程序或下半部处理程序，则返回非 0

6.6 通用 IRQ 层

内核使用一个通用 IRQ 层（Generic IRQ Layer，即 GenIRQ）进行中断的处理。

GenIRQ 的目的是为设备驱动程序提供有关中断处理过程的完整抽象，从而在设备驱动执行注册、激活、禁止以及释放中断等操作的同时，不必了解关于硬件的任何细节，因此不需要进行代码修改即可适用于不同的平台。

6.6.1 GenIRQ 的起源及发展

在 GenIRQ 出现之前，对于中断的处理，内核的实现是使用一个超级的服务程序__do_IRQ 来完成所有的操作。

而在 GenIRQ 最初的实现中，中断控制器的硬件操作细节被封装在 hw_interrupt_type 结构中，__do_IRQ 函数则致力于完成对 IRQ 流程（IRQ Flow）的处理，只是在需要对硬件进行操作的时候，调用相应 hw_interrupt_type 结构对象的成员函数，比如 ack()、end()等。

于是，原有的超级处理函数__do_IRQ 就被划分为 IRQ Flow 处理与中断控制器硬件操作两个层次。

但是，并不是所有类型的中断都具有同样的 IRQ Flow，比如电平触发中断和边沿触发中断的 IRQ Flow 就截然不同，如果希望在同一个函数中都进行支持，必然会导致混乱的代码逻辑。

同时，同一个中断控制器也可能支持不同的中断类型，比如 IOAPIC 既支持电平触发又可支持边沿触发，因此它具有两个 hw_interrupt_type 结构对象：ioapic_level_type 以及 ioapic_edge_type，它们的底层硬件操作方式大致相同，区别主要体现在电平触发中断和边沿触发中断的 IRQ Flow，这样就导致了很多不必要的重复代码。

为了解决这些问题，Thomas Gleixner 和 Ingo Molnar 在 2.6.17 版内核提交了 GenIRQ 补丁，对 GenIRQ 的抽象层次进行更为清晰的划分。

6.6.2 GenIRQ 的抽象层次

如图 6.13 所示，在最新的内核代码里，GenIRQ 共抽象为 3 层：中断控制器硬件操作、IRQ 流程处理以及驱动 API。

驱动API

- -

IRQ 流程处理

- -

中断控制器硬件操作

图 6.13　通用 IRQ 层的抽象层次

1. 中断控制器硬件操作

新的 irq_chip 结构取代了旧的 hw_interrupt_type 结构，并增加了一些新的底层硬件操作成员函数，包括 mask、mask_ack、unmask、set_type 以及 set_wake。基于 irq_chip 结构的众多成员函数，内核实现了对中断控制器更为精细的管理。

2. IRQ 流程处理

在 irq_desc 结构中，irq_chip 指针取代了旧的 hw_interrupt_type 指针，同时添加了一个新的函数指针 handle_irq，指向处理该 IRQ 线的实际服务程序，这样就摒弃了之前对所有中断都采用一个服务程序 __do_IRQ 的做法。

中断之间最大的不同在于如何产生和被处理，GenIRQ 定义了 4 种 IRQ Flow 类型。

Level-triggered（电平触发）中断，只要设备驱动了（asserts）中断线就会被激活。它们在处理时必须被屏蔽，并仅仅在设备放弃中断线之后被 unmask。

Edge-triggered（边沿触发）中断，在中断线发生改变时产生，如从低电压到高电压，或者从高电压到低电压。它们在处理时不需要被屏蔽，但如果没有被屏蔽，在前一个中断处理结束之前，更多的中断将到达。因此，内核必须跟踪正在 pending 的中断，中断服务程序必须循环处理所有的中断。

Simple 中断不需要任何特别的控制，能够直接被处理。

Per CPU 中断被绑定于单个 CPU 上。它们类似于 Simple 中断，甚至更加简单，因为中断服务程序仅仅运行在同一 CPU 上，不需要使用任何锁机制。

之前的内核尝试在一个函数中处理上面所有的 4 种情况，而新的代码创建了针对不同 IRQ Flow 类型的服务程序，然后将它们指定给各个 irq_desc 的 handle_irq 指针，因此代码能够根据特定的需要进行优化。

3. 驱动 API

在驱动 API 层，GenIRQ 的变化相当少，因此原有的驱动一般不需要做出改变即可使用。但仍然有一些新的特点需要被了解。

（1）有一些新增标志用于 request_irq 函数。

SA_TRIGGER_LOW 和 SA_TRIGGER_HIGH：用于电平触发方式的中断，指示中断是发生在低电平还是高电平。

SA_TRIGGER_FALLING 和 SA_TRIGGER_RISING：用于边沿触发方式的中断。

（2）新增如下函数用于直接更改一个 IRQ 的 Flow 类型。

```
int set_irq_type(unsigned int irq, unsigned int type);
```

type 可以取 IRQ_TYPE_EDGE_RISING、IRQ_TYPE_EDGE_FALLING、IRQ_TYPE_EDGE_BOTH、IRQ_TYPE_LEVEL_HIGH、IRQ_TYPE_LEVEL_LOW、IRQ_TYPE_SIMPLE 和 IRQ_TYPE_PERCPU 之中的一个。

（3）某些设备可以产生将系统从 suspend 状态唤醒的中断，比如通过 keypad 中断唤醒系统。GenIRQ 增加了下面的函数来激活或者禁止中断控制器的这个特点：

```
int set_irq_wake(unsigned int irq, unsigned int on);
```

6.7 多处理器系统中的中断处理

华清远见

多处理器系统正在变得越来越普通，它在带来速度、性能等好处的同时，也带来了新的问题和新的思考。

6.7.1 处理器间中断

在多处理器系统中，操作系统需要在多个处理器间协调操作，这通常是通过处理器间中断（Inter-Processor Interrupt，IPI）实现的。

IPI 是一种特殊的硬件中断，由处理器发出，被其他处理器接收，以便于处理器间通信或同步。

通常并不明确区分 IPI 与设备中断，当一个处理器接收到一个中断时，如果发现另一个处理器处理该中断更加合理，则可以通过 IPI 机制将该中断传递到其他的处理器，实现处理器的负载平衡。

当一个 CPU 想对另一个 CPU 发送中断信号时，就在自己的本地 APIC 的 ICR（Interrupt Command Register，中断命令寄存器）中存放其中断向量，和目标 CPU 拥有的本地 APIC 的标识，触发中断。IPI 中断信号经由 APIC 总线传递到目标 APIC，那个收到中断的 APIC 就向自己所属的 CPU 发送一个中断。

Linux 针对 IA32 的 SMP 系统定义了 5 种 IPI，中断向量号分别为 251～255。

（1）CALL_FUNCTION_VECTOR。发往自己除外的所有 CPU，强制它们执行指定的函数。

（2）RESCHEDULE_VECTOR。使被中断的 CPU 重新调度。

（3）INVLIDATE_TLB_VECTOR。使被中断的 CPU 废弃自己的 TLB 缓存内容。

（4）ERROR_APIC_VECTOR。错误的 APIC 向量，应该从不发生。

（5）SPUROUS_APIC_VECTOR。假的 APIC 向量，应该从不发生。

6.7.2 中断亲和力

中断亲和力（SMP IRQ affinity）是指将一个或多个中断服务程序绑定到特定的 CPU 上运行，内核自 2.4 版本开始对此提供支持。

中断亲和力可以通过操作/proc/irq 目录下的文件来控制。对于已经注册中断服务程序的硬件设备，在/proc/irq 目录下存在一个以该中断号命名的目录，该目录下有一个 smp_affinity 文件（SMP 体系结构才有该文件），它是一个 CPU 的位掩码，其中的每一位对应于一个 CPU，可以用来设置该中断的亲和力，默认值为 0xffffffff，表明把中断发送到所有的 CPU 上去处理。如果中断控制器不支持 IRQ affinity，则不能改变此默认值。注意不能将其设置为 0x0，即关闭所有 CPU 对该中断的处理。

以网卡（eth1，中断号 44）为例，在具有 8 个 CPU 的服务器上来设置网卡中断的亲和力（参见文件 Documentation\IRQ-affinity.txt）。

```
[root@moon 44]# cat smp_affinity
ffffffff
[root@moon 44]# echo 0f > smp_affinity
[root@moon 44]# cat smp_affinity
0000000f
[root@moon 44]# ping -f h
PING hell (195.4.7.3): 56 data bytes
...
--- hell ping statistics ---
6029 packets transmitted, 6027 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.1/0.4 ms
[root@moon 44]# cat /proc/interrupts | grep 44:
 44:  0    1785    1785    1783    1783    1   1    0   IO-APIC-level  eth1
[root@moon 44]# echo f0 > smp_affinity
[root@moon 44]# ping -f h
PING hell (195.4.7.3): 56 data bytes
..
```

```
--- hell ping statistics ---
2779 packets transmitted, 2777 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.5/585.4 ms
[root@moon 44]# cat /proc/interrupts | grep 44:
 44:  1068  1785  1785  1784  1784   1069   1070    1069   IO-APIC-level  eth1
[root@moon 44]#
```

首先只允许在 CPU 0～3 上处理网卡中断，接着运行 ping 程序，不难发现在 CPU 4～7 上并没有对网卡中断进行处理。然后只在 CPU 4～7 上对网卡中断进行处理，CPU 0～3 不对网卡中断进行任何处理，运行 ping 程序之后，再次查看/proc/interrupts 文件时，不难发现 CPU 4～7 上的中断次数明显增加，而 CPU 0～3 上的中断次数没有太大的变化。

利用中断亲和力，我们可以在多处理系统中均衡各个 CPU 的负载，比如对于具有多个网卡的 SMP 系统，可以将各个网卡的中断信号分别绑定到不同的 CPU 上，从而可以大幅提高系统能够处理的网络通信流量。

6.7.3 中断负载均衡

中断负载均衡是指将重负载 CPU 上的中断迁移到较空闲的 CPU 上进行处理。如果希望系统支持这个功能，必须在编译内核的时候配置 CONFIG_IRQBALANCE 选项。

中断负载均衡的实现代码位于 arch\i386\kernel\io-apic.c 文件。其中的 balanced_irq_init 函数会进行中断负载均衡模块的初始化，同时会创建一个内核线程 kirqd 用于执行具体的均衡处理。代码如下：

```
666 static int __init balanced_irq_init(void)
667 {
        ……
703     printk(KERN_INFO "Starting balanced_irq\n");
704     if (!IS_ERR(kthread_run(balanced_irq, NULL, "kirqd")))
705         return 0;
706     printk(KERN_ERR "balanced_irq_init: failed to spawn balanced_irq");
        ……
715 }
```

而 kirqd 每隔 5s 调用一次 do_irq_balance 函数，进行中断的迁徙。

6.8 中断的下半部

到目前为止，对中断的处理过程似乎已经很完备了：设备根据自己的需要产生中断信号，将其通过中断控制器发送给 CPU，CPU 检测到该中断信号后停止它当前的工作，然后根据中断向量号获取中断服务程序的地址并执行。

如果设备的请求都很容易满足，那么这样的处理过程确实已经足够简单高效，并且无可挑剔，既节省了内核轮询时耗费的大量资源，又不会影响系统的正常运作。但现实总是那么不尽如人意，内核往往需要完成大量的工作才能满足设备的需要。

比如对于一个网卡中断，网卡的目的是，内核收到中断后将网卡收到的网络数据进行处理，这个处理的完整过程就应该是：从网卡硬件的缓冲区中把网卡收到的网络数据包复制到系统内存中，然后对这个数据包进行 TCP/IP 协议栈的处理，首先是链路层，然后是 IP 层（包括分片、奇偶校验等），再然后是 TCP 层（TCP 层的实现相当复杂，会花费比较长的时间对数据包进行一些状态或者内容的分析处理），最后会通过 socket 将数据包传入用户空间。

在将数据传入用户空间之前，所有这些将花费大量时间的处理都会在中断服务程序中完成。在这段时间内，CPU 将不再响应网卡发来的其他中断，网卡将因为自身缓存的不足而丢失数据。

为了解决这种响应一次中断需要完成大量工作的问题，Linux 将对中断的处理划分为两个部分：上半部和下半部。上半部是实际响应中断的程序，也就是传统意义上的中断服务程序，只完成一些比较紧急和必要的功能，比如对中断的到达进行响应确认。

而下半部则是完成其他那些可以延缓处理的部分，在下半部的执行期间，所有的中断都是打开的。这样上半部的执行将非常快，且允许在下半部工作期间，上半部仍能继续服务新的中断。

对于上面网卡中断的例子来说，上半部将只是完成将数据复制到内存的工作，而其他的协议栈处理等不是特别紧急的工作则是放到了下半部去处理。

6.8.1 下半部的实现机制

内核最初通过 BH（bottom half）机制实现下半部，定义了一个函数指针数组，共有 32 个函数指针，采用数组索引来访问与之相对应的一套函数（也就是 bottom half）。

同时内核定义了一组枚举变量，包含了这个数组中多数位置所对应的意义。并约定某个驱动使用某个 bottom half 位置，比如串口中断就约定使用 SERIAL_BH。

```
enum
{
    TIMER_BH = 0,
    TQUEUE_BH,
    DIGI_BH,
    SERIAL_BH,
    RISCOM8_BH,
    ……
};
```

显而易见，原始的 BH 机制有个很大的局限，就是个数限制在 32 个以内，因为每个 bottom half 上只能挂接一个函数，随着系统硬件越来越多，这个数目显然是不够用的。另外，同一时间不允许任何两个 bottom half 同时运行，即使它们分属不同的 CPU。

因此，内核引入了任务队列（task queue）来取代 BH 机制，每个队列中都包含一个由延缓执行的函数组成的链表。内核使用 struct tq_struct 描述延缓执行的任务，使用时驱动可以使用 DECLARE_TASK_QUEUE 宏定义自己的任务队列，然后定义一个 struct tq_struct 变量，并将其注册到自己定义的任务队列上，之后可以通过手工调用 run_task_queue()函数启动该任务对列中的所有任务，也可以直接将 run_task_queue()的地址作为 BH 机制中函数指针数组中的一个函数指针，进而让 BH 机制在合适的时候自动执行延缓的任务。

但大多数情况下，驱动都没有必要定义自己的任务队列，因为系统已经预定义了一些任务队列，比如 tq_disk，由内存管理模块使用。

但是任务队列的机制仍然不够灵活，并不能满足一些对性能要求比较高的子系统，比如网络。因此内核又引入了软中断（softirq）和 tasklet 的机制，并在 2.6 内核中彻底摒弃了 BH 机制，同时任务队列也被工作队列（work queue）所取代。

1. 软中断

这里所说的软中断与系统调用所使用软中断（确切地说应该是软件中断）并不是同一个概念。软中断沿用了最早的 BH 机制的思想，但在这个 BH 机制之上，实现了一个更加庞大和复杂的软中断子系统。

软中断在编译期间静态分配，内核定义了一个软中断数组，最多可以包含 32 个软中断（由 struct softirq_action 描述），每个被注册的软中断都占据该数组中的一项，但与 BH 机制相比，它们可以在所有的 CPU 上同时执行，即使是两个相同类型的软中断，所以使用时需要特别小心。同时，内核还定义了一组枚举变量，预定义了几个软中断的用途。

```
enum
{
    HI_SOFTIRQ=0,        // 优先级高的 tasklets
    TIMER_SOFTIRQ,   // 定时器的下半部
    NET_TX_SOFTIRQ,  // 发送网络数据包
```

```
        NET_RX_SOFTIRQ,  // 接收网络数据包
        BLOCK_SOFTIRQ,
        TASKLET_SOFTIRQ, // tasklets
        SCHED_SOFTIRQ,
    #ifdef CONFIG_HIGH_RES_TIMERS
        HRTIMER_SOFTIRQ,
    #endif
    };
```

一个软中断只能被中断服务程序抢占，而不会去抢占其他软中断。通常中断服务程序会在返回前触发它的软中断以使其在稍后被执行，然后内核在执行完该中断服务程序后，就会马上调用 do_softirq()函数，执行软中断去完成剩余的任务。

2．tasklet

tasklet 是利用软中断实现的一种下半部机制。与软中断相比，两个不同类型的 tasklet 可以在不同的 CPU 上同时执行，但相同类型的 tasklet 却不能够同时执行，因此 tasklet 的锁保护要求较低，这就大大降低了驱动开发的难度。

在 tasklet 的源码注释中还说明了几点特性，总之归结为一点，就是：同一个 tasklet 只会在一个 CPU 上运行。

tasklet 本身就是软中断，可以是 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 两者之一，它们之间的唯一区别就是 HI_SOFTIRQ 类型的软中断要比 TASKLET_SOFTIRQ 类型的高。当内核调度一个 tasklet 时，就会唤起这两个软中断中的一个，随后该软中断会被特定的函数进行处理，执行所有已调度的 tasklet。

3．工作队列

工作队列是另一种实现下半部的机制，它在 2.6 内核被引入用来取代任务队列。与前面讨论的软中断和 tasklet 机制不同的是，工作队列把延缓的工作交由一个内核线程去执行，因此通过工作队列执行的代码是运行在进程上下文的，允许重新调度和睡眠。

另外，你可以通过工作队列将工作延后到一个明确的时间间隔后再执行，因此它通常用来处理不是很紧急的事件。

6.8.2 下半部机制的选择

通过上节的讨论，我们知道下半部可以通过 5 种方式实现：BH 机制、任务队列、软中断、tasklet、工作队列。其中，前两种在 2.6 内核中已经看不到了，因此可供我们选择的其实就是后三种。

软中断和 tasklet 运行于中断上下文，工作队列靠内核线程实现，运行于进程上下文。因此，如果需要将任务推迟到进程上下文完成，工作队列是必然的选择，否则软中断和 tasklet 可能要更合适，因为使用工作队列时不可避免要带来到上下文切换的消耗。

相对于软中断而言，tasklet 对锁的要求要低得多。如果要使用下半部的模块代码本身对于保护考虑的不够好，那么选择 tasklet 就比较有必要了。

如果从易用性考虑，要首推工作队列，其次是 tasklet，最后才是软中断，因为它必须要静态的创建。简单地说，关于下半部的使用，一般的驱动程序开发者需要做出的选择是：需要一个可调度的实体来执行需要推后完成的工作，即有休眠的需要吗？要是有，工作队列就是唯一的选择，否则最好用 tasklet，如果必须专注于性能的提高，那么就考虑软中断吧。

华清远见

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

华清远见