Linux 内核中断内幕

苏 春艳, 在读研究生
杨 小华 (normalnotebook@126.com), 在读研究生

简介： 本文对中断系统进行了全面的分析与探讨，主要包括中断控制器、中断分类、中断亲和力、中断线程化与 SMP 中的中断迁徙等。首先对中断工作原理进行了简要分析，接着详细探讨了中断亲和力的实现原理，最后对中断线程化与非线程化中断之间的实现机理进行了对比分析。

什么是中断

Linux 内核需要对连接到计算机上的所有硬件设备进行管理，毫无疑问这是它的份内事。如果要管理这些设备，首先得和它们互相通信才行，一般有两种方案可实现这种功能：

1. 轮询（polling） 让内核定期对设备的状态进行查询，然后做出相应的处理；
2. 中断（interrupt） 让硬件在需要的时候向内核发出信号（变内核主动为硬件主动）。

第一种方案会让内核做不少的无用功，因为轮询总会周期性的重复执行，大量地耗用 CPU 时间，因此效率及其低下，所以一般都是采用第二种方案 。[注释 1]

从物理学的角度看，中断是一种电信号，由硬件设备产生，并直接送入中断控制器（如 8259A）的输入引脚上，然后再由中断控制器向处理器发送相应的信号。处理器一经检测到该信号，便中断自己当前正在处理的工作，转而去处理中断。此后，处理器会通知 OS 已经产生中断。这样，OS 就可以对这个中断进行适当的处理。不同的设备对应的中断不同，而每个中断都通过一个唯一的数字标识，这些值通常被称为中断请求线。

回页首

APIC vs 8259A

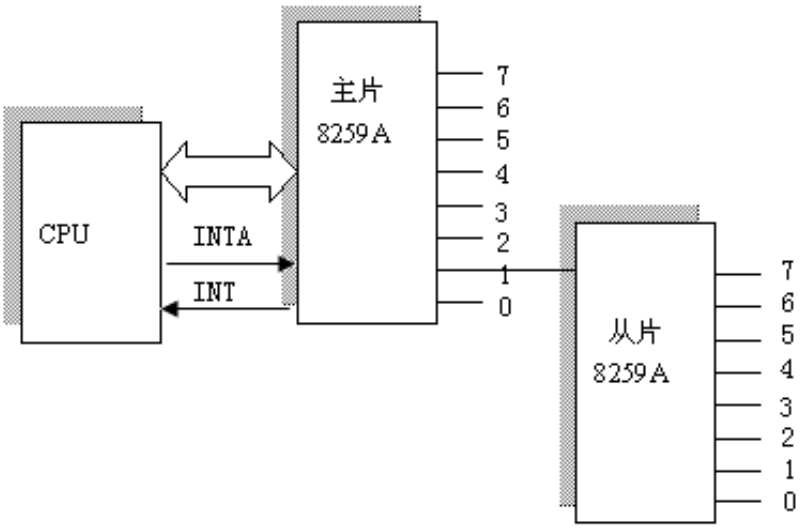
X86计算机的 CPU 为中断只提供了两条外接引脚：NMI 和 INTR。其中 NMI 是不可屏蔽中断，它通常用于电源掉电和物理存储器奇偶校验；INTR是可屏蔽中断，可以通过设置中断屏蔽位来进行中断屏蔽，它主要用于接受外部硬件的中断信号，这些信号由中断控制器传递给 CPU。

常见的中断控制器有两种：

1. 可编程中断控制器8259A

传统的 PIC（Programmable Interrupt Controller）是由两片 8259A 风格的外部芯片以“级联”的方式连接在一起。每个芯片可处理多达 8 个不同的 IRQ。因为从 PIC 的 INT 输出线连接到主 PIC 的 IRQ2 引脚，所以可用 IRQ 线的个数达到 15 个，如图 1 所示。

图 1：8259A 级联原理图

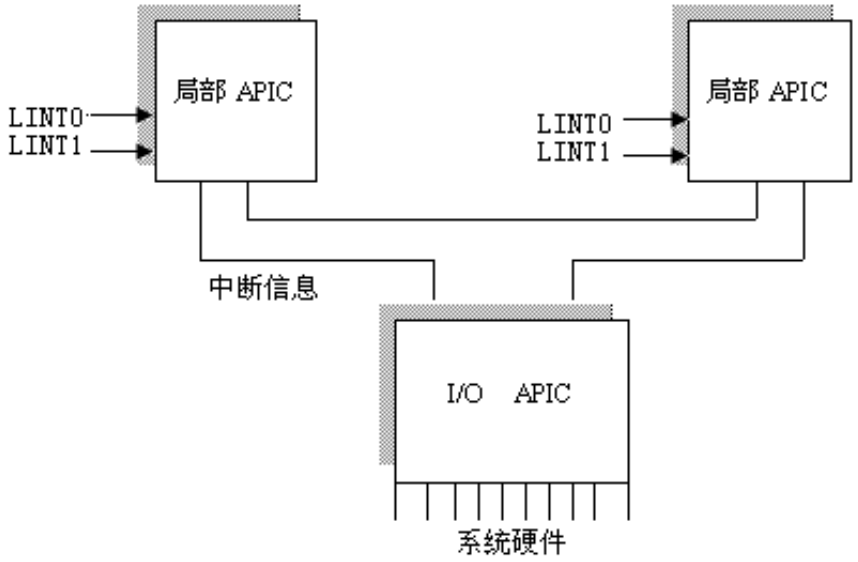


2. 高级可编程中断控制器（APIC）

8259A 只适合单 CPU 的情况，为了充分挖掘 SMP 体系结构的并行性，能够把中断传递给系统中的每个 CPU 至关重要。基于此理由，Intel 引入了一种名为 I/O 高级可编程控制器的新组件，来替代老式的 8259A 可编程中断控制器。该组件包含两大组成部分：一是“本地 APIC”，主要负责传递中断信号到指定的处理器；举例来说，一台具有三个处理器的机器，则它必须相对的要有三个本地 APIC。另外一个重要的部分是 I/O APIC，主要是收集来自 I/O 装置的 Interrupt 信号且在当那些装置需要中断时发送信号到本地 APIC，系统中最多可拥有 8 个 I/O APIC。

每个本地 APIC 都有 32 位的寄存器，一个内部时钟，一个本地定时设备以及为本地中断保留的两条额外的 IRQ 线 LINT0 和 LINT1。所有本地 APIC 都连接到 I/O APIC，形成一个多级 APIC 系统，如图 2 所示。

图 2：多级I/O APIC系统



目前大部分单处理器系统都包含一个 I/O APIC 芯片，可以通过以下两种方式来对这种芯片进行配置：

1) 作为一种标准的 8259A 工作方式。本地 APIC 被禁止，外部 I/O APIC 连接到 CPU，两条 LINT0 和 LINT1 分别连接到 INTR 和 NMI 引脚。

2) 作为一种标准外部 I/O APIC。本地 APIC 被激活，且所有的外部中断都通过 I/O APIC 接收。

辨别一个系统是否正在使用 I/O APIC，可以在命令行输入如下命令：

```
# cat /proc/interrupts
           CPU0
  0:      90504    IO-APIC-edge  timer
  1:        131    IO-APIC-edge  i8042
  8:          4    IO-APIC-edge  rtc
  9:          0    IO-APIC-level  acpi
 12:        111    IO-APIC-edge  i8042
 14:       1862    IO-APIC-edge  ide0
 15:         28    IO-APIC-edge  ide1
177:          9    IO-APIC-level  eth0
185:          0    IO-APIC-level  via82cxxx
...
```

如果输出结果中列出了 IO-APIC，说明您的系统正在使用 APIC。如果看到 XT-PIC，意味着您的系统正在使用 8259A 芯片。

回页首

中断分类

中断可分为同步（synchronous）中断和异步（asynchronous）中断：

1. 同步中断是当指令执行时由 CPU 控制单元产生，之所以称为同步，是因为只有在一条指令执行完毕后 CPU 才会发出中断，而不是发生在代码指令执行期间，比如系统调用。

2. 异步中断是指由其他硬件设备依照 CPU 时钟信号随机产生，即意味着中断能够在指令之间发生，例如键盘中断。

根据 Intel 官方资料，同步中断称为异常（exception），异步中断被称为中断（interrupt）。

中断可分为可屏蔽中断（Maskable interrupt）和非屏蔽中断（Nomaskable interrupt）。异常可分为故障（fault）、陷阱（trap）、终止（abort）三类。

从广义上讲，中断可分为四类：中断、故障、陷阱、终止。这些类别之间的异同点请参看 表 1。

表 1：中断类别及其行为

类别	原因	异步/同步	返回行为
中断	来自I/O设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	返回到当前指令
终止	不可恢复的错误	同步	不会返回

X86 体系结构的每个中断都被赋予一个唯一的编号或者向量（8 位无符号整数）。非屏蔽中断和异常向量是固定的，而可屏蔽中断向量可以通过对中断控制器的编程来改变。

回页首

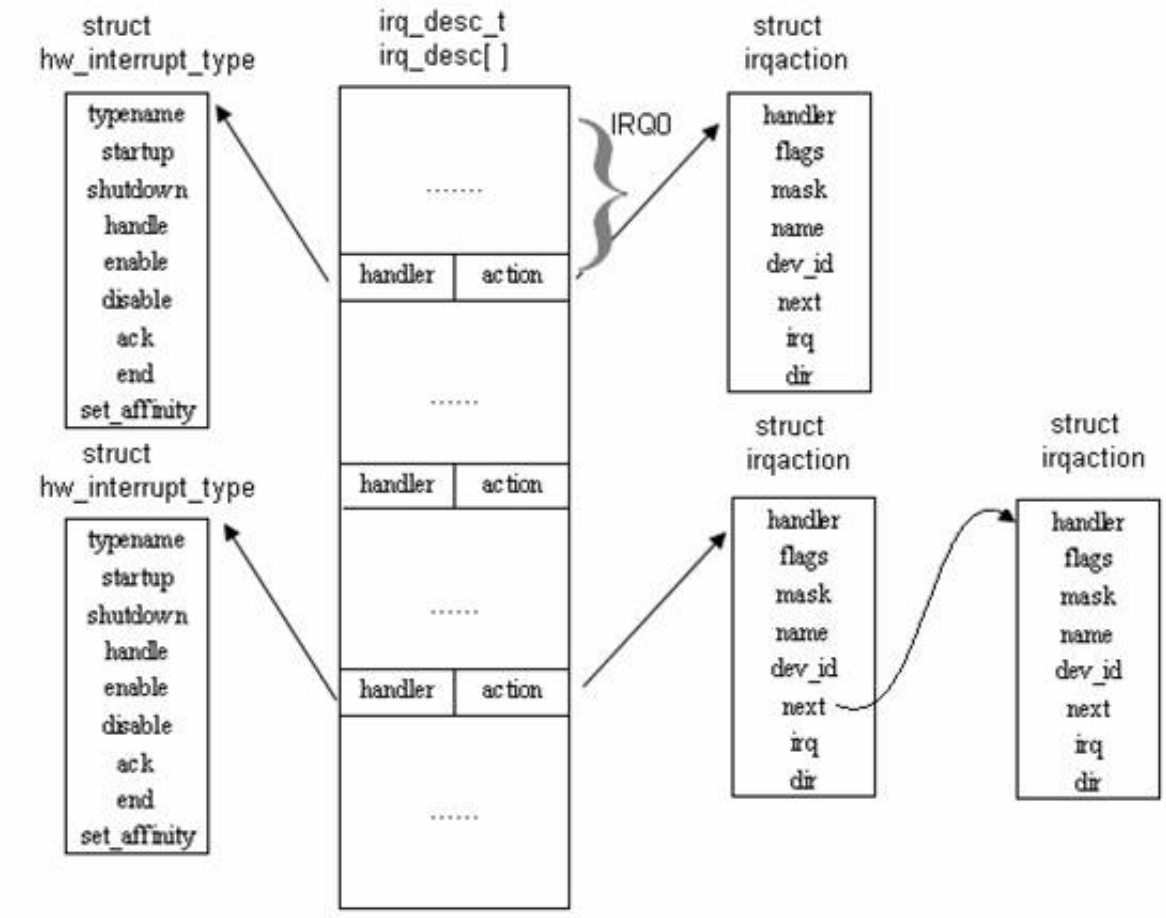
Linux 2.6 中断处理原理简介

中断描述符表（Interrupt Descriptor Table，IDT）是一个系统表，它与每一个中断或异常向量相联系，每一个向量在表中存放的是相应的中断或异常处理程序的入口地址。内核在允许中断发生前，也就是在系统初始化时，必须把 IDT 表的初始化地址装载到 idtr 寄存器中，初始化表中的每一项。

当处于实模式下时，IDT 被初始化并由 BIOS 程序所使用。然而，一旦 Linux 开始接管，IDT 就被移到 ARM 的另一个区域，并进行第二次初始化，因为 Linux 不使用任何 BIOS 程序，而使用自己专门的中断服务程序（例程）（interrupt service routine，ISR）。中断和异常处理程序很像常规的 C 函数

有三个主要的数据结构包含了与 IRQ 相关的所有信息：hw_interrupt_type、irq_desc_t 和 irqaction，图3 解释了它们之间是如何关联的。

图 3：IRQ 结构之间的关系



在 X86 系统中，对于 8259A 和 I/O APIC 这两种不同类型的中断控制器，hw_interrupt_type 结构体被赋予不同的值，具体区别参见表 2。

表 2：8259A 和 I/O APIC PIC 的区别

8259A	I/O APIC
	static struct hw_interrupt_type ioapic_edge_type = { .typename = "IO-APIC-edge", .startup =
	startup_edge_ioapic, .shutdown

```
                    = shutdown_edge_ioapic,
                    .enable = enable_edge_ioapic,
                    .disable =
static struct       disable_edge_ioapic, .ack =
hw_interrupt_type   ack_edge_ioapic, .end =
i8259A_irq_type = {end_edge_ioapic, .set_affinity
"XT-PIC",           = set_ioapic_affinity, };
startup_8259A_irq,  static struct
shutdown_8259A_irq, hw_interrupt_type
enable_8259A_irq,   ioapic_level_type = {
disable_8259A_irq,  .typename = "IO-APIC-level",
mask_and_ack_8259A, .startup =
end_8259A_irq, NULL startup_level_ioapic,
};                  .shutdown =
                    shutdown_level_ioapic, .enable
                    = enable_level_ioapic,
                    .disable =
                    disable_level_ioapic, .ack =
                    mask_and_ack_level_ioapic,
                    .end = end_level_ioapic,
                    .set_affinity =
                    set_ioapic_affinity, };
```

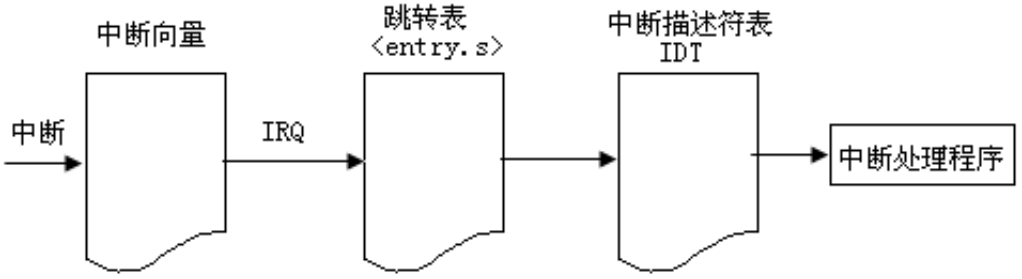
在中断初始化阶段，调用 hw_interrupt_type 类型的变量初始化 irq_desc_t 结构中的 handle 成员。在早期的系统中使用级联的8259A，所以将用 i8259A_irq_type 来进行初始化，而对于SMP系统来说，要么以 ioapic_edge_type，或以 ioapic_level_type 来初始化 handle 变量。

对于每一个外设，要么以静态（声明为 static 类型的全局变量）或动态（调用 request_irq 函数）的方式向 Linux 内核注册中断处理程序。不管以何种方式注册，都会声明或分配一块 irqaction 结构（其中 handler 指向中断服务程序），然后调用 setup_irq() 函数，将 irq_desc_t 和 irqaction 联系起来。

当中断发生时，通过中断描述符表 IDT 获取中断服务程序入口地址，对于 32≤ i ≤255（i≠128） 之间的中断向量，将会执行 push $i-256,jmp common_interrupt 指令。随之将调用 do_IRQ() 函数，以中断向量为 irq_desc[] 结构的下标，获取 action 的指针，然后调用 handler 所指向的中断服务程序。

从以上描述，我们不难看出整个中断的流程，如图 4 所示：

图 4：X86中断流



本文作者之一曾经对2.6.10的中断系统进行过情景分析，有兴趣的读者可以和作者取得联系，获取相关资料。

回页首

中断绑定——中断亲和力（IRQ Affinity）

在 SMP 体系结构中，我们可以通过调用系统调用和一组相关的宏来设置 CPU 亲和力（CPU affinity），将一个或多个进程绑定到一个或多个处理器上运行。中断在这方面也毫不示弱，也具有相同的特性。中断亲和力是指将一个或多个中断源绑定到特定的 CPU 上运行。中断亲和力最初由 Ingo Molnar 设计并实现。

在 /proc/irq 目录中，对于已经注册中断处理程序的硬件设备，都会在该目录下存在一个以该中断号命名的目录 IRQ# ，IRQ# 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，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,不能改变此默认值，同时也不能关闭所有的 CPU 位掩码，即不能设置成 0x0。

我们以网卡（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（以下数据出自内核源码 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]#
```

在上例中，我们首先只允许在 CPU0~3 上处理网卡中断，接着运行 ping 程序，不难发现在 CPU4~7 上并没有对网卡中断进行处理。然后只在 CPU4~7 上对网卡中断进行处理， CPU0~3 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 CPU4~7 上的中断次数明显增加，而 CPU0~3 上的中断次数没有太大的变化。

在探讨中断亲和力的实现原理之前，我们首先来了解 I/O APIC 中的组成。

I/O APIC 由一组 24 条 IRQ 线，一张 24 项的中断重定向表（Interrupt Redirection Table），可编程寄存器，以及通过 APIC 总线发送和接收 APIC 信息的一个信息单元组成。其中与中断亲和力息息相关的是中断重定向表， 中断重定向表表中的每一项都可以被单独编程以指明中断向量和优先级、目标处理器及选择处理器的方式 。

通过表 2，不难发现 8259A 和 APIC 中断控制器最大不同点在于 hw_interrupt_type 类型变量的最后一项。对于 8259A 类型，set_affinity 被置为 NULL，而对于 SMP 的 APIC 类

型，set_affinity 被赋值为 set_ioapic_affinity。

在系统初始化期间，对于 SMP 体系结构，将会调用 setup_IO_APIC_irqs() 函数来初始化 I/O APIC 芯片，芯片中的中断重定向表的 24 项被填充。在系统启动期间，所有的 CPU 都执行 setup_local_APIC() 函数，完成本地的 APIC 初始化。当有中断被触发时，将相应的中断重定向表中的值转换成一条消息，然后，通过 APIC 总线把消息发送给一个或多个本地 APIC 单元，这样，中断就能立即被传递给一个特定的 CPU，或一组 CPU，或所有的 CPU，从而来实现中断亲和力。

当我们通过 cat 命令将 CPU 掩码写进 smp_affinity 文件时，此时的调用路线图为：write() ->sys_write() ->vfs_write() ->proc_file_write() ->irq_affinity_write_proc() ->set_affinity() ->set_ioapic_affinity() ->set_ioapic_affinity_irq() ->io_apic_write()；其中在调用 set_ioapic_affinity_irq() 函数时，以中断号和 CPU 掩码作为参数，接着继续调用 io_apic_write()，修改相应的中断重定向中的值，来完成中断亲和力的设置。当执行 ping 命令时，网卡中断被触发，产生了一个中断信号，多 APIC 系统根据中断重定向表中的值，依照仲裁机制，选择 CPU0~3 中的某一个 CPU，并将该信号传递给相应的本地 APIC，本地 APIC 又中断它的 CPU，整个事件不通报给其他所有的 CPU。

回页首

新特性展望——中断线程化（Interrupt Threads）

在嵌入式领域，业界对 Linux 实时性的呼声越来越高，对中断进行改造势在必行。在 Linux 中，中断具有最高的优先级。不论在任何时刻，只要产生中断事件，内核将立即执行相应的中断处理程序，等到所有挂起的中断和软中断处理完毕后才能执行正常的任务，因此有可能造成实时任务得不到及时的处理。中断线程化之后，中断将作为内核线程运行而且被赋予不同的实时优先级，实时任务可以有比中断线程更高的优先级。这样，具有最高优先级的实时任务就能得到优先处理，即使在严重负载下仍有实时性保证。

目前较新的 Linux 2.6.17 还不支持中断线程化。但由 Ingo Molnar 设计并实现的实时补丁，实现了中断线程化。最新的下载地址为：

http://people.redhat.com/~mingo/realtime-preempt/patch-2.6.17-rt9

下面将对中断线程化进行简要分析。

在初始化阶段，中断线程化的中断初始化与常规中断初始化大体上相同，在 start_kernel() 函数中都调用了 trap_init() 和 init_IRQ() 两个函数来初始化 irq_desc_t 结构体，不同点主要体现在内核初始化创建 init 线程时，中断线程化的中断在 init() 函数中还将调用 init_hardirqs（kernel/irq/manage.c（已经打过上文提到的补丁）），来为每一个 IRQ 创建一个内核线程，最高实时优先级为 50，依次类推直到 25，因此任何 IRQ 线程的最低实时优先级为 25。

```
void __init init_hardirqs(void)
{
……
        for (i = 0; i < NR_IRQS; i++) {
                irq_desc_t *desc = irq_desc + i;
                if (desc->action && !(desc->status & IRQ_NODELAY))
                        desc->thread = kthread_create(do_irqd, desc, "IRQ %d", irq);
    ……
        }
}
static int do_irqd(void * __desc)
{
    ……
```

```
        /*
         * Scale irq thread priorities from prio 50 to prio 25
         */
        param.sched_priority = curr_irq_prio;
        if (param.sched_priority > 25)
                curr_irq_prio = param.sched_priority - 1;
    ……
}
```

如果某个中断号状态位中的 IRQ_NODELAY 被置位，那么该中断不能被线程化。

在中断处理阶段，两者之间的异同点主要体现在：两者相同的部分是当发生中断时，CPU 将调用 do_IRQ() 函数来处理相应的中断，do_IRQ() 在做了必要的相关处理之后调用 __do_IRQ()。两者最大的不同点体现在 __do_IRQ() 函数中，在该函数中，将判断该中断是否已经被线程化（如果中断描述符的状态字段不包含 IRQ_NODELAY 标志，则说明该中断被线程化了），对于没有线程化的中断，将直接调用 handle_IRQ_event() 函数来处理。

```
fastcall notrace unsigned int __do_IRQ(unsigned int irq, struct pt_regs *regs)
{
……
        if (redirect_hardirq(desc))
                goto out_no_end;
……
action_ret = handle_IRQ_event(irq, regs, action);
……
}
int redirect_hardirq(struct irq_desc *desc)
{
……
        if (!hardirq_preemption || (desc->status & IRQ_NODELAY) || !desc->thread)
                return 0;
……
        if (desc->thread && desc->thread->state != TASK_RUNNING)
                wake_up_process(desc->thread);
……
}
```

对于已经线程化的情况，调用 wake_up_process() 函数唤醒中断处理线程，并开始运行，内核线程将调用 do_hardirq() 来处理相应的中断，该函数将判断是否有中断需要被处理，如果有就调用 handle_IRQ_event() 来处理。handle_IRQ_event() 将直接调用相应的中断处理函数来完成中断处理。

不难看出，不管是线程化还是非线程化的中断，最终都会执行 handle_IRQ_event() 函数来调用相应的中断处理函数，只是线程化的中断处理函数是在内核线程中执行的。

并不是所有的中断都可以被线程化，比如时钟中断，主要用来维护系统时间以及定时器等，其中定时器是操作系统的脉搏，一旦被线程化，就有可能被挂起，这样后果将不堪设想，所以不应当被线程化。如果某个中断需要被实时处理，它可以像时钟中断那样，用 SA_NODELAY 标志来声明自己非线程化，例如：

```
static struct irqaction irq0 = {
        timer_interrupt, SA_INTERRUPT | SA_NODELAY, CPU_MASK_NONE, "timer", NULL, NULL
};
```

其中，SA_NODELAY 到 IRQ_NODELAY 之间的转换，是在 setup_irq() 函数中完成的。

回页首

中断负载均衡—SMP体系结构下的中断

中断负载均衡的实现主要封装在 arch\ arch\i386\kernel\io-apic.c 文件中。如果在编译内核时配置了 CONFIG_IRQBALANCE 选项，那么 SMP 体系结构中的中断负载均衡将以模块的形式存在于内核中。

```
late_initcall(balanced_irq_init);
#define late_initcall(fn)               module_init(fn)  //include\linux\init.h
```

在 balanced_irq_init() 函数中，将创建一个内核线程来负责中断负载均衡：

```
static int __init balanced_irq_init(void)
{   ……
        printk(KERN_INFO "Starting balanced_irq\n");
        if (kernel_thread(balanced_irq, NULL, CLONE_KERNEL) >= 0)
                return 0;
        else
                printk(KERN_ERR "balanced_irq_init: failed to spawn balanced_irq");
    ……
}
```

在 balanced_irq() 函数中，每隔 5HZ=5s 的时间，将调用一次 do_irq_balance() 函数，进行中断的迁徙。将重负载 CPU 上的中断迁移到较空闲的CPU上进行处理。

回页首

总结

随着中断亲和力和中断线程化的相继实现，Linux 内核在 SMP 和实时性能方面的表现越来越让人满意，完全有理由相信，在不久的将来，中断线程化将被合并到基线版本中。本文对中断线程化的分析只是起一个抛砖引玉的作用，当新特性发布时，不至于让人感到迷茫。

- 注释 1：轮询也不是毫无用处，比如NAPI，就是轮询与中断相结合的经典案例。

参考资料

1. Rebert Love，《Linux Kernel Development，2rd Edition》，机械工业出版社，2006。
2. Daniel P. Bovet，Marco Cesati，《Understanding the Linux Kernel，3rd Edition》，东南大学出版社，2006。
3. Jonatban Corbet 等，魏永明等译，《Linux设备驱动程序》，中国电力出版社，2006。
4. Gordon Fischer 等，《The Linux Kernel Prime》，机械工业出版社，2006。

- developerWorks 中国网站 Linux 技术专区

作者简介

苏春艳：在读研究生，主要在Linux系统下从事嵌入式开发。

杨小华，目前从事 Linux 内核方面的研究，喜欢捣鼓 Linux 系统，对 Linux 中断系统比较了解。可以通过 normalnotebook@126.com与他取得联系。