对中断的一点思考

2006-2-26

声明

如果本文有什么错误，请大家指正，谢谢！发信至：YCSEVOLI@gmail.com

对于 X86 的单处理器机器，一般采用可编程中断控制器 8259A 做为中断控制电路。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 输入线。因为从 PIC 的 INT 输出线连接到主 PIC 的 IRQ2 引脚，所以可用 IRQ 线的个数限制为 15，如图 1 所示。

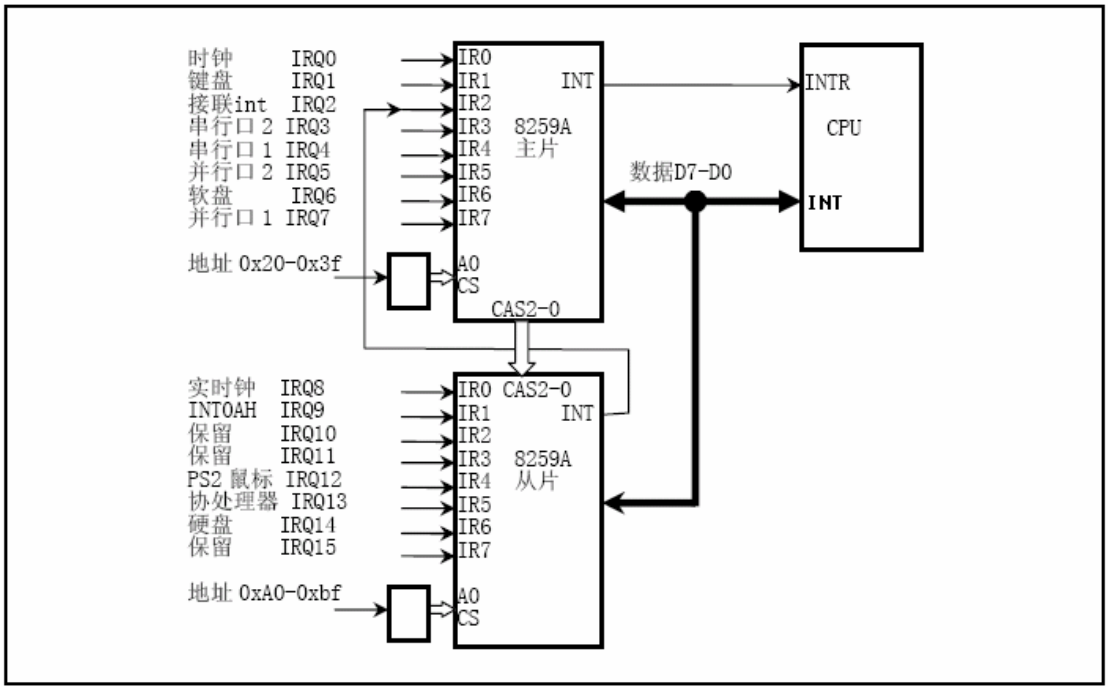

图 1 8259A 级联原理图（此图摘自《Linux 内核完全注释》）

“中断屏蔽寄存器”（*Interrupt Mask Register*，简称 IMR）用于屏蔽 8259A 的中断信号输入，每一位对应一个输入。当 IMR 中的 bit[i](0≤i≤7)位被置 1 时，相对应的中断信号输入线 IRi 上的中断信号将被 8259A 所屏蔽，也即 IRi 被禁止。

当外设产生中断信号时（由低到高的跳变信号，80x86 系统中的 8259A 是边缘触发的，*Edge Triggered*），中断信号被输入到“中断请求寄存器”（*Interrupt Request Register*，简称 IRR），并同时看看 IMR 中的相应位是否已被设置。如果没有被设置，则 IRR 中的相应位被设置为 1，表示外设产生一个中断请求，等待 CPU 服务。

然后，8259A 的优先级仲裁部分从 IRR 中选出一个优先级最高中断请求。优先级仲裁之后，8259A 就通过其 INT 引脚向 CPU 发出中断信号，以通知 CPU 有外设请求中断服务。CPU 在其当前指令执行完后就通过他的 INTA 引脚给 8259A 发出中断应答信号，以告诉 8259A，CPU 已经检测到有中断信号产生。

8259A 在收到 CPU 的 INTA 信号后，将优先级最高的那个中断请求在 ISR 寄存器（*In－Service Registe*r，简称 ISR）中对应的 bit 置 1，表示该中断请求已得到 CPU 的服务，同时 IRR 寄存器中的相应位被清零重置。

然后，CPU 再向 8259A 发出一个 INTA 脉冲信号，8259A 在收到 CPU 的第二个 INTA 信号后，将中断请求对应的中断向量放到数据总线上，以供 CPU 读取。CPU 读到中断向量后，就可以装入执行相应的中断处理程序。

如果 8259A 工作在 AEOI（*Auto End Of Interrupt*，简称 AEOI）模式下，则当他收到 CPU

的第二个 INTA 信号时，它就自动重置 ISR 寄存器中的相应位。否则，ISR 寄存器中的相应位就一直保持为 1，直到 8259A 显示地收到来自于 CPU 的 EOI 命令。

打住，各位看官读到这里，能回答如下问题吗？

1. 在执行中断处理程序时，中断一直是关闭着的吗？
2. 在执行中断处理程序时，本条中断线上的中断是否会被屏蔽？
3. 如果该条中断线被屏蔽了，那么是否一直要到该中断返回（即执行 iter 指令）时才开启？
4. 禁止中断后，异常还会执行吗？timer 还会被执行吗？

如果不能回答这些问题，请继续欣赏。如果你能回答，请关闭本文档，努力工作吧，或拿起一本英语书看看，这年头不好混，多看看英语吧 ！:）

当中断发生，CPU 在穿越中断门时会关闭本处理器上所有的中断。此时中断执行路线是：common_interrupt->do_IRQ()->__do_IRQ()->handle_IRQ_event()->具体的中断处理程序。大家都知道中断类型包括三种：

标志	含义
SA_INTERRUPT	当该位被设置时，表明这是一个快速的中断处理程序。在本地处理器上，快速中断处理程序在禁止所有中断的情况下运行。除了时钟中断外，绝大多数中断都不使用该标志。
SA_SHIRQ	该位表示中断可以在设备之间共享。
SA_SAMPLE_RANDOM	该位指出产生的中断能对/dev/random 设备和/dev/urandom 设备使用的熵池有贡献。

表 1 中断类型标志位及其含义表

如果相应的中断处理程序在注册时，即调用 request_irq（）函数进行中断处理程序注册时，会传递这三种中类型中的一个或数个。如果没有指定 SA_INTERRUPT 该类型。则在 handle_IRQ_event（）函数中会第一次开中断；如果指定了该参数，则会在关闭中断的情况下执行中断处理程序。

```
fastcall int handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
                              struct irqaction *action)
{
    int ret, retval = 0, status = 0;
    if (!(action->flags & SA_INTERRUPT))
        local_irq_enable();
   ……//调用中断处理程序
    local_irq_disable();
    return retval;
}
```

如果此时开中断了，本条 IRQ 线上的中断也打开了吗？我要告诉你的是，在执行到这里的时候，本条线上的中断已经被屏蔽了，但也不是问题 3 中所说的一直到 iret 时才打开。

```
fastcall unsigned int __do_IRQ(unsigned int irq, struct pt_regs *regs)
{
        irq_desc_t *desc = irq_desc + irq;
        ……
        desc->handler->ack(irq);
        ……
        action_ret = handle_IRQ_event(irq, regs, desc->action);
        ……
        desc->handler->end(irq);
        ……
}
```

desc->handler->ack(irq);最终会调用 mask_and_ack_8259A()(为什么会调用该函数，请查看源代码或相关书籍)。mask_and_ack_8259A 的功能是：因为中断处理器在将中断请求“上报”到 CPU 后，期待 CPU 给它一个确认（ACK），也就是给 8259A 芯片一个应答信号，表示“我已经在处理”，应答时应该遵循这样的顺序：首先，屏蔽相应的 IRQ；然后，发送 EOI（*End of Interrupt*）命令。此外，如果 IRQ 来自于从 8259A，还必须先向从 8259A 发送 EOI 命令，再向主 8259A 发送 EOI 命令。如果 IRQ 来自于主 8259A，则仅仅向主 8259A 发送 EOI 命令就可以了。当一个中断服务结束后，CPU 可利用中断结束命令 EOI 通知 8259A，以便复位 ISR 中的相应位。因此当调用 handle_IRQ_event（）时，即使开中断，该条中断线的中断也是关闭的。一直到调用 desc->handler->end(irq);时才清除中断屏蔽。

细心的读者可能还有一个问题，为什么在 handle_IRQ_event（）返回时，还要关闭本地所有的中断(即代码中的 local_irq_disable();)。因为在中断返回时，将会进行退栈清理性的工作，如果此时响应中断，鬼知道后果是什么？嘿嘿， Linus Torvalds 肯定知道结果。因为他比鬼还厉害 :）

虽然，中断关闭了，但异常并没有关闭。关中断只是关掉了外部中断，cli 只是设置 EFLAGS 寄存器的 IF 位，如果该位被清除，则表示 CPU 会禁止外部中断传递信号给 INTR 引脚，但对于 CPU 内部异常和不可屏蔽中断（NMI）并不起作用。由于 timer 是外部中断，所以在禁中断后，timer 中断将不在起作用，一直到开中断。

咔，提问时间到了，该进入另一个环节了！

广告时间，以上 sentence 由《春运帝国》赞助播映！！！

1. 如果在 desc->handler->ack(irq);和 desc->handler->end(irq);之间，该条中断线上再次发生中断，该中断是否会被丢失？
2. 在中断处理快结束时，会执行软中断。可在执行 do_softirq()时，又会执行

   if (in_interrupt())

   return;

   难道软中断不在中断中吗？岂不是执行到这里都返回了？

如果你又知道这些问题的答案，那我只能自认倒霉了，难道你就是传说中的 hacker!!!:).

对于第一个问题，我也不能给出明确的答案。我只是把所收集的资料写出来。至于对不对，有大家自己去判断。

中断不会丢失，会被挂起（pending），保存在 IPR（*Interrupt Pending Register*）中，等到中断允许位打开后再执行相应的中断处理程序。（*但我查了 8259A 的寄存器，好像没有这个寄存器。*）

在《源代码情景分析》一书中 p216 第 10 行提到：“这样，就把本来可能发生在同一通道（甚至可能来自同一中断源）的中断嵌套化解为一个循环”。本人认为这种嵌套仅仅针对的是 SMP 的情况。因为对单 CPU 来说，ack 操作已经将本条中断线给屏蔽了，根本不可能再响应了。

如果哪位这里有比较好的权威性的答案，请记得发封邮件给我，先谢过了。

在 do_IRQ()中，将会调用 irq_enter（）()和 irq_exit（）两个函数。

```
fastcall unsigned int do_IRQ(struct pt_regs *regs)
{
    ……
    irq_enter();
    ……
    __do_IRQ(irq, regs);
    ……
    irq_exit();
    ……
}
```

在执行 irq_enter（）时，将会调用(preempt_count() += HARDIRQ_OFFSET)；在执行 irq_exit（）函数时，又会：

```
void irq_exit(void)
{
    preempt_count()-= IRQ_EXIT_OFFSET;// # define IRQ_EXIT_OFFSET    HARDIRQ_OFFSET
    if (!in_interrupt() && local_softirq_pending())
        do_softirq();
    preempt_enable_no_resched();
}
```

而 in_interrupt()函数只是对 preempt_count 的某些位进行测试。所以在执行 do_softirq（）时，preempt_count 经过一次加减运算后，将值还原了。所以只要不是中断嵌套的话，in_interrupt()会为假。所以执行到这里时，根本不会返回，除非中断嵌套。

我们也该休息一下了，广告之后马上回来，进入下一个环节－－有关调度的问题。

我曾经在一个培训资料上看到如下的结论：

实时应用中，中断的发生不但要求迅速的中断服务，还要求迅速的调度有关的进程进入运行，在用户空间中对事件处理。 可是，如果这样的中断发生在内核时，本次中断返回是不会引起调度的，而要到最初使

CPU 从用户空间进入内核的那次系统调用或中断（或异常）返回时才会发生调度。倘若内核中的这段代码恰好需要较长时间来完成的话，或者连续又发生几次中断的话，就可能将调度过分的推迟。

当中断发生在内核时，本次中断返回时是有可能引起调度的，原因是，只要 need_resched 被置位并且 preempt_count＝0，如果这这两个条件都满足，那么就会发生 schedule（）操作，我认为写这段话的作者，当时可能研究的是 2.4 的内核，因为那时的内核是不可抢占的。当返回到用户态时，只需要对 need_resched 进行检查。

另外一个问题：

当 idle 运行时，发生外部中断 A，中断处理程序 A 将一个进程 P1 唤醒，并设置了调度标志 need_resched，在中断处理程序 A 还没有结束前，又有一高优先级的外部中断 B 发生，响应 B，当中断处理程序 B 结束后，内核进行调度，选择中断 A 所唤醒的进程 P1 运行，此后，产生许多可运行的进程，致使 idle 不能很快再运行。这不就存在着很大的问题，甚者会导致该条线上的中断永远都不能响应。

的确不错，当调用 wake_up_process（）时，如果被唤醒进程的优先级比当前进程的优先级高，那么将设置 need_resched 位。但这个问题忽略了一个很重要的问题，那就是中断 B 返回时，当返回内核态时（我想不可能返回到用户态吧），将会对 need_resched 和 preempt_count 进行检查，如果这两者都满足，才会进行调度。从上面分析可知，当发生中断嵌套时，preempt_count 此时等于一个很大的值，虽然 B 在执行时，preempt_count 经过一次加和减的操作，但 A 还是将该值设置成禁止抢占的，所以中断 B 返回时，根本不可能发生调度，而是会执行中断上下文 A。

说明：有些结论来自论坛上网友的回答，向他们表示感谢。

友情鸣谢：

[1] www.linuxforum.net

[2] Daniel P.Bovel & Marco Cesati 《深入理解 Linux 内核》 中国电力出版社，2004

[3] 毛德操，胡希明 《Linux 内核源代码情景分析》 浙江大学出版社，2001

[4] Robert Love 《Linux 内核设计与实现》 机械工业出版社 ，2004

超值附送中断返回流程图：