1.总体介绍

这个接口一般是 kernel 自己使用的，用于内核子系统里面的处理，比如网络还有 tasklet 的处理，要用到这个。

已知的软中断类型有：

enum

{

HI_SOFTIRQ=0, 高优先级 tasklet

TIMER_SOFTIRQ,

NET_TX_SOFTIRQ,

NET_RX_SOFTIRQ,

BLOCK_SOFTIRQ,

BLOCK_IOPOLL_SOFTIRQ,

TASKLET_SOFTIRQ, 普通优先级 tasklet

SCHED_SOFTIRQ,

HRTIMER_SOFTIRQ,

RCU_SOFTIRQ, /* Preferable RCU should always be the last softirq */

NR_SOFTIRQS

};

驱动的编写是不会直接用到这个 softirq 的。

struct softirq_action{} 表示一个软中断处理函数

open_softirq() 注册一个软中断。

raise_softirq() 触发一个软中断，内部会 wakeup_softirqd 来真正唤起 softirq 的 handler 来处理。

有关软中断到底执行在中断上下文还是进程上下文的问题：

1.如果负载较轻，则执行在中断上下文。但注意，这个中断上下文，和真正的硬中断上下文有别，体现在内核的测试上下文的接口打印如下：

in_interrupt: yes | in_irq: no | in_softirq: yes | in_serving_softirq: yes

2.如果负载较重，则会调度 ksoftirqd 来帮忙，此时就是执行在进程上下文了。

所以不能绝对的说 softirq 就一定是中断上下文。

软中断在一个 cpu 上是串行执行的。

2.raise_softirq()逻辑

```
/*
 * This function must run with irqs disabled!
 */
inline void raise_softirq_irqoff(unsigned int nr)
{
    __raise_softirq_irqoff(nr);

    /*
     * If we're in an interrupt or softirq, we're done
     * (this also catches softirq-disabled code). We will
     * actually run the softirq once we return from
     * the irq or softirq.
     *
     * Otherwise we wake up ksoftirqd to make sure we
     * schedule the softirq soon.
     */
    if (!in_interrupt())
        wakeup_softirqd();
}

void raise_softirq(unsigned int nr)
{
    unsigned long flags;

    local_irq_save(flags);
    raise_softirq_irqoff(nr);
    local_irq_restore(flags);
}
```

可以看到，先关中断，调用 local_irq_save 关中断，然后调用 raise_softirq_irqoff->__raise_softirq_irqoff->or_softirq_pending

```
void __raise_softirq_irqoff(unsigned int nr)
{
    lockdep_assert_irqs_disabled();
    trace_softirq_raise(nr);
    or_softirq_pending(1UL << nr);
}
```

```
#define or_softirq_pending(x)   (__this_cpu_or(local_softirq_pending_ref, (x)))
```

```
#define local_softirq_pending_ref irq_stat.__softirq_pending
```

内部 or_softirq_pending 就是或操作了 per cpu 变量 irq_stat.__softirq_pending

in_interrupt 展开下如下：

```
#define in_interrupt()      (irq_count())
```

```
#define irq_count() (preempt_count() & (HARDIRQ_MASK | SOFTIRQ_MASK \
                 | NMI_MASK))
```

然后 in_interrupt 判断如果不是在中断上下文，则调用 wakeup_softirqd()唤醒 softirqd 线程来处理软中断（进程上下文）

```
static void wakeup_softirqd(void)
{
    /* Interrupts are disabled: no need to stop preemption */
    struct task_struct *tsk = __this_cpu_read(ksoftirqd);

    if (tsk && tsk->state != TASK_RUNNING)
        wake_up_process(tsk);
}
```

如果 in_interrupt 判断在中断上下文，则因为刚刚已经置位 irq_stat.__softirq_pending，在中断返回的时候自然会进行软中断的处理：

从__handle_domain_irq 开始看：

```
int __handle_domain_irq(struct irq_domain *domain, unsigned int hwirq,
            bool lookup, struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);
    unsigned int irq = hwirq;
    int ret = 0;

    irq_enter();

#ifdef CONFIG_IRQ_DOMAIN
    if (lookup)
        irq = irq_find_mapping(domain, hwirq);
#endif

    /*
     * Some hardware gives randomly wrong interrupts.  Rather
     * than crashing, do something sensible.
     */
    if (unlikely(!irq || irq >= nr_irqs)) {
        ack_bad_irq(irq);
        ret = -EINVAL;
    } else {
        generic_handle_irq(irq);
    }

    irq_exit();
    set_irq_regs(old_regs);
    return ret;
} « end __handle_domain_irq »
```

```
void irq_exit(void)
{
    __irq_exit_rcu();
    rcu_irq_exit();
     /* must be last! */
    lockdep_hardirq_exit();
}
```

```
static inline void __irq_exit_rcu(void)
{
#ifndef __ARCH_IRQ_EXIT_IRQS_DISABLED
    local_irq_disable();
#else
    lockdep_assert_irqs_disabled();
#endif
    account_irq_exit_time(current);
    preempt_count_sub(HARDIRQ_OFFSET);
    if (!in_interrupt() && local_softirq_pending())
        invoke_softirq();

    tick_irq_exit();
}
```

可以看到，__irq_exit_rcu 里面，先做了 preempt_count_sub(HARDIRQ_OFFSET)，所以如果不考虑中断嵌套，那么下一步判断 in_interrupt 结果就是 0

因为 preempt 计数减过了，再加上有 local_softirq_pending 条件为 1，所以执行 invoke_softirq()。

注 1：如果中断是嵌套执行的，虽然现在 kernel 都是关嵌套的，则不急着执行 invoke_softirq()，等真正中断退出的时候再做 invoke_softirq()）

注 2：因为 in_interrupt 其实查询的是硬中断+软中断+NMI 中断，所以如果当前正在软中断上下文，一样不要急着做 invoke_softirq()，从而保证了在一个 CPU 上的 softirq 是串行执行的（注意：多个 CPU 上还是有可能并发的）。

```
static inline void invoke_softirq(void)
{
    if (ksoftirqd_running(local_softirq_pending()))
        return;

    if (!force_irqthreads) {
#ifdef CONFIG_HAVE_IRQ_EXIT_ON_IRQ_STACK
        /*
         * We can safely execute softirq on the current stack if
         * it is the irq stack, because it should be near empty
         * at this stage.
         */
        __do_softirq();
#else
        /*
         * Otherwise, irq_exit() is called on the task stack that can
         * be potentially deep already. So call softirq in its own stack
         * to prevent from any overrun.
         */
        do_softirq_own_stack();
#endif
    } else {
        wakeup_softirqd();
    }
} « end invoke_softirq »
```

在 invoke_softirq 中，如果当前 ksoftirqd 已经在处理软中断，说明现在有人在处理，直接 return。

如果 force_irqthreads 也就是强制中断线程化是 1，则调用 wakeup_softirqd 也就是用线程处理，否则：

根据是不是要在 irq 的栈上执行软中断，分别执行__do_softirq 或者 do_softirq_own_stack。

以__do_softirq()为例：

```
asmlinkage __visible void __softirq_entry __do_softirq(void)
{
    unsigned long end = jiffies + MAX_SOFTIRQ_TIME;
    unsigned long old_flags = current->flags;
    int max_restart = MAX_SOFTIRQ_RESTART;
    struct softirq_action *h;
    bool in_hardirq;
    __u32 pending;
    int softirq_bit;

    /*
     * Mask out PF_MEMALLOC as the current task context is borrowed for the
     * softirq. A softirq handled, such as network RX, might set PF_MEMALLOC
     * again if the socket is related to swapping.
     */
    current->flags &= ~PF_MEMALLOC;

    pending = local_softirq_pending();
    account_irq_enter_time(current);

    __local_bh_disable_ip(_RET_IP_, SOFTIRQ_OFFSET);
    in_hardirq = lockdep_softirq_start();

restart:
    /* Reset the pending bitmask before enabling irqs */
    set_softirq_pending(0);

    local_irq_enable();

    h = softirq_vec;

    while ((softirq_bit = ffs(pending))) {
        unsigned int vec_nr;
        int prev_count;

        h += softirq_bit - 1;

        vec_nr = h - softirq_vec;
        prev_count = preempt_count();

        kstat_incr_softirqs_this_cpu(vec_nr);

        trace_softirq_entry(vec_nr);
        h->action(h);
        trace_softirq_exit(vec_nr);

        if (unlikely(prev_count != preempt_count())) {
            pr_err("huh, entered softirq %u %s %p with preempt_count %08x, exited with %08x?\n",
                   vec_nr, softirq_to_name[vec_nr], h->action,
                   prev_count, preempt_count());
            preempt_count_set(prev_count);
        }
        h++;
        pending >>= softirq_bit;
    } » end while (softirq_bit=ffs(pend... »

    if (__this_cpu_read(ksoftirqd) == current)
        rcu_softirq_qs();
    local_irq_disable();

    pending = local_softirq_pending();
    if (pending) {
        if (time_before(jiffies, end) && !need_resched() &&
            --max_restart)
            goto restart;

        wakeup_softirqd();
    }

    lockdep_softirq_end(in_hardirq);
    account_irq_exit_time(current);
    __local_bh_enable(SOFTIRQ_OFFSET);
    WARN_ON_ONCE(in_interrupt());
    current_restore_flags(old_flags, PF_MEMALLOC);
} « end __do_softirq »
```

以上逻辑大致为：

取出当前 local_softirq_pending

while 循环逐一处理 local_softirq_pending 的 bit，每个 bit 都用 softirq_action 的 action()函数处理

退出 while 循环后，再次检查 local_softirq_pending，如果有，代表刚刚 while 循环处理期间有新增的 pending，且没有超过 timeout，。跳转到 while 循环再跑一遍。

超过 timeout 的，调用 wakeup_softirqd 延后处理。