你应该知道Linux内核softirq

From:写代码的篮球球痴 嵌入式Linux Yesterday

说起这个softirq，很多人还是一头雾水，觉得这个是什么东西，**跟tasklets和workqueue**有什么不同。

每次谈到这个，很多人，包括我，都是有点紧张，特别是面试的时候，因为你一旦说错了什么，那么你这次面试估计就歇菜了。

谈到这个，我们不得不说中断，中断处理，我相信很多人都是知道的，中断分为上半部和下半部，原来的Linux 内核是没有下半部的，中断来了，就在中断里处理事件，说白了，就是执行一些函数操作，但是这个会导致一个问题，就是系统调度慢了，对于用户来说，你用手机的时候，感觉到十分卡顿，真想摔了这个手机，所以，后来就出现了中断下半部。

中断下半部的机制有很多种。例如：softirq、tasklet、workqueue或是直接创建一个kernel thread来执行bottom half（这在旧的**kernel**驱动中常见，现在，一个理智的**driver**厂商是不会这么做的）

tasklet 是基于softirq实现的，所以我们讨论tasklet的时候，其实也是在说softirq，他们都是运行在中断上下文的。

workqueue和softirq不同的是，它运行是进程上下文。

为什么需要这么多机制来实现中断下半部呢？

你可以理解，我如果要去纽约，我可以坐飞机，可以坐轮船，也可以开小汽车，甚至，我还可以骑自行车，中断下半部也是一样，tasklet的出现，是因为把softirq封装起来，更加方便别人使用。workqueue的话，紧急程度就没有softirq那么紧急，可以说优先级没有那么高，如果是非常紧急的事情，比如网络事件，我们还是优先使用softirq来实现。

Linux 内核里面可以有多少种softirq呢？

softirq是一种非常紧急的事件，所以说，不是你想用就用的，内核里面定义了一个枚举变量来说明softirq支持的类型。

```
/* PLEASE, avoid to allocate new softirqs, if you need not _really_ high
   frequency threaded job scheduling. For almost all the purposes
   tasklets are more than enough. F.e. all serial device BHs et
   al. should be converted to tasklets, not to softirqs.
 */

enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        BLOCK_IOPOLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ, /* Unused, but kept as tools rely on the
                            numbering. Sigh! */
        RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

        NR_SOFTIRQS
};
```

看到上面的那段英文了没，它说，「大部分情况下，**tasklets**已经满足你的要求，就不要只想着用**softirq**」

软中断是什么时候执行的呢？

软中断和workqueue存在非常大的区别，我们在上面说过，软中断是在中断上下文的，但是中断上半部已经脱离了中断了，它如何跟中断上下文存在千丝万缕的联系呢？说到这里，我们不拿出代码来说，那就是耍流氓了。

```
/*
 * Exit an interrupt context. Process softirqs if needed and possible:
 */
void irq_exit(void)
{
#ifndef __ARCH_IRQ_EXIT_IRQS_DISABLED
       local_irq_disable();
#else
       WARN_ON_ONCE(!irqs_disabled());
#endif

       account_irq_exit_time(current);
       preempt_count_sub(HARDIRQ_OFFSET);
       if (!in_interrupt() && local_softirq_pending())
//此处判断是否从中断上下文退出，并判断是否有软中断任务挂起待处理
              invoke_softirq();
//启用，排程软中断处理

       tick_irq_exit();
       rcu_irq_exit();
       trace_hardirq_exit(); /* must be last! */
}
```

我们看看irq_exit()这个函数，这个函数是在什么时候调用？就是中断上半部执行结束后，需要跳出中断上半部的时候，我们需要执行irq_exit()，然后在里面有一个判断。

```
if (!in_interrupt() && local_softirq_pending())
              invoke_softirq();

       tick_irq_exit();
       rcu_irq_exit();
       trace_hardirq_exit(); /* must be last! */
}
```

这几行代码就是判断当前是否需要执行软中断，说白了，就是CPU需要执行一段优先级非常高的代码，这段代码需要在中断结束，关闭中断后马上执行。

```
#define local_softirq_pending() this_cpu_read(irq_stat.__softirq_pending)
```

那我们在中断上半部执行结束后，如何设置需要执行softirq呢？使用这个函数

```
#define set_softirq_pending(x) __this_cpu_write(irq_stat.__softirq_pending, (x))
```

softirq相关的代码都在 softirq.c 这个文件里面，如果想有更深入了解的同学，可以一睹源码风采，不吹，C语言的源码真是一个宝藏，细细评味，可以挖掘出非常多的好东西。

我们分析下 invoke_softirq

这个函数是执行softirq的函数，里面也有一些判断的东西，我看了下源码，理解了下，顺便解读下自己的看法，如果有疑问或者问题的，请读者们指出来，只有不断的探讨，大家才可能收获更多的东西。

```
static inline void invoke_softirq(void)
{
	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();
	}
}
```

不会看注释的码农不是好码农，不写注释的码农就不是码农，如果你想写代码，就一定要会写注释，同时，你也要会看别人的注释，好吧，这类源码都是老外写的，所以我们一定要习惯看英文注释，慢慢看，不要着急，学会上下文推敲，这样才像一个大神嘛。

我不是大神，所以，我就瞎说一下上面使用一个 force_irqthreads 来区分一个东西，就是软中断上下文，软中断上下文是不能睡眠的，你知道的，你要是在中断里面睡眠，那系统调度就起不来了，起不来的原因那可真是五花八门，因为你不知道进入中断的时候做了什么事情，在中断里面的时候，我们只有更高优先级的中断才能打断当前中断，现在新版本的Linux内核取消了中断嵌套，那你要是在中断里面睡觉，就没有人叫你起床了，那就只能出现panic，挂机了。用我的话来说，那就是睡死了。

如果你的softirq里面执行很多东西，在软中断上文没有执行完，那你就需要用到软中断线程把剩下的事情做完，然后就出现了wakeup_softirqd()，这个就是处理软中断下半部的。

看看__do_softirq()里面做的事情吧

```
asmlinkage __visible void __softirq_entry __do_softirq(void)
{
    unsigned long end = jiffies + MAX_SOFTIRQ_TIME; //最大处理时间：2毫秒
    unsigned long old_flags = current->flags;
    int max_restart = MAX_SOFTIRQ_RESTART; //最大回圈次数：10次
    struct softirq_action *h;
    bool in_hardirq;
    __u32 pending;
    int softirq_bit;

    /*
     * Mask out PF_MEMALLOC s 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 swap
     */
    current->flags &= ~PF_MEMALLOC;

    pending = local_softirq_pending(); //获取本地CPU上等待处理的软中断掩码
    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); //清除本地CPU上等待处理的软中断掩码

    local_irq_enable(); // 开中断状态下处理软中断

    h = softirq_vec; // h指向软中断处理函式阵列首元素

    while ((softirq_bit = ffs(pending))) { //依次处理软中断，软中断编号越小，越优先处理，优先顺
        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); //呼叫软中断回拨处理函式
            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 %0
                       vec_nr, softirq_to_name[vec_nr], h->action,
                       prev_count, preempt_count());
                preempt_count_set(prev_count);
            }

            //回圈下一个等待处理的软中断
            h++;
            pending >>= softirq_bit;
        }

        rcu_bh_qs();
        local_irq_disable(); //关中断，判断在处理上次软中断期间，硬中断处理函式是否又排程了软中断

        pending = local_softirq_pending();
        if (pending) { //软中断再次被排程
            if (time_before(jiffies, end) && !need_resched() &&
                --max_restart) //没有达到超时时间，也不需要被排程，并且排程次数也没有超过10次
                goto restart; //重新执行软中断

            wakeup_softirqd(); //否则唤醒软中断核心执行绪处理剩下的软中断，当前CPU退出软中断上下文
        }

        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);
    }
```

这段代码主要是展示了软中断上下文的处理策略，一次处理所有等待的软中断，处理轮训结束或者时间超过2ms，就跳出软中断上下文，跑到软中断线程里面去执行，避免系统响应过慢。

如何加入新的软中断？

说了那么多，这个应该是重点了，一直强调，不要加软中断，使用tasklet就够了，但是无法避免就是有人要用啊。

内核使用下面函数来新增一个软中断

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
```

```
{
        softirq_vec[nr].action = action;
}
```

当然了，你可以新增一个枚举变量

内核里面是这样使用下面这个函数调用

```
open_softirq(RCU_SOFTIRQ, rcu_process_callbacks);
```

好了，就说这么多~

右下角，你懂的ʕ◔ϖ◔ʔ

————————END————————

扫码或长按关注

回复「 **加群** 」进入技术群聊