

# Linux 设备驱动之中断上半部、下半部

—编写者:草根老师(程姚根)

我们知道在操作系统中，任何时间都会有中断产生，一旦产生中断必定要去执行中断处理函数。不论什么时候，我们都应该让中断处理函数执行的时间尽可能短。还记得我们在通过request\_irq注册一个中断的时候，其中一个中断标志叫IRQF\_DISABLED。我们都知道如果有这个标志，中断处理函数在执行的时候，是屏蔽外界一切中断的。如果中断处理函数执行的时间长，是一件很危险的时间，因为此时其它硬件请求的中断都不能被响应。

到这里，你可能会问两个问题：

## (1)不使用IRQF\_DISABLED标志不就可以了吗？

我们一般使用这个标志的最主要原因是想让我们的中断处理函数能不受干扰的尽快的执行完。但是很多时候，都会事如愿违，因为我们总是很贪婪，想在中断处理函数中干很多事情。

还有一点，就算我们不是使用这个标志，其它的中断虽然可以产生，但是同一个中断号的中断此时是被屏蔽的。

## (2)既然此时同一个中断号的中断是屏蔽的，我只需要在中断处理函数中，重新开中断，一切问题不就解决了吗？

嗯，理论上来说是可以的。但是这个时候会发生中断嵌套的，会让问题变的更复杂。

## 那怎么办？是不是Linux 系统有相应的机制来解决这些问题呢？

是的，在Linux 系统中，将中断处理函数分为了两部分来执行。上半部，在执行的时候是屏蔽中断的。下半部在执行的时候，中断是不屏蔽的即可以被中断打断。这样我们就可以把比较紧急的事情放在上半部执行，不是很紧急的事情推倒下半部去执行。

此处又有几个问题诞生：

## (1)什么是上半部，什么是下半部？

嗯，咱们直白一点。正常情况下，一个中断对应一个中断处理函数。分成两部分之后，相当于，一个中断对应了两个函数，一个先执行，一个后执行。先执行的叫做上半部(执行的时候不可被打断)，后执行的叫做下半部(执行的时候可以被打断)。

## (2)哪些是紧急的事情，哪些不是紧急的事情？

这个问题就是仁者见仁，智者见智，需要程序员自己去把控了。把控了好，一切顺风顺水，如果把控的不好，可能会导致操作系统的效率降低。

当然，江湖大侠也有一些经验，大家可以根据自己的实际情况做出选择。

<1>如果一个任务时间十分敏感，将其放在上半部

<2>如果一个任务和硬件有关，将其放在上半部

<3>如果一个任务要保证不被其他中断打断，将其放在上半部

<4>其他所有任务，考虑放在下半部

### (3)下半部是什么时候执行？

大神都说"**Linux 内核会在合适的时机执行下半部**"。我知道此处你有问题，但请保留，现在还没到回到问题的时间。你只需知道，Linux 内核会找合适的时机执行下半部。

### (4)Linux 内核是如何实现下半部的呢？

在整个Linux 内核版本中，下半部的实现机制有很多。但是在2.6版本之后，用的比较多有三种:软中断，tasklet，workqueue。好了，下面我们就一起来看看这三种机制是如何使用的吧!

## 一、软中断

一提这个名字，很多初学者就会有无数的问题抛出来。不急，听我慢慢道来。

**这里软中断不是一种中断，更不是ARM五中异常模式中的软中断异常,要知道ARM的软中断异常是通过"swi"指令触发的。**

那它是什么呢？

**它是Linux 内核实现中断下半部的一种机制。或者你可以认为，它是通过软件的形式对硬件中断的一种模拟。**

嗯，Linux 内核是如何模拟的呢?我们一起来看一下。

### (1)硬件中断在Linux 内核中是有中断号的，软中断有吗？

在软中断的世界里，也有软中断号，但是我们更喜欢叫做软中断类型。2.6.35的Linux 内核支持10种软中断类型，如下:

```
/* 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.
```



从上图大家可以清晰的看到,Linux 支持的10中软中断，软中断号越小优先级越高，优先得到处理。

这些软中断号都已经被别人使用了，如果我想注册自己的软中断怎么办呀？

做法很简单，大家只需在Linux 内核源码树下找到include/linux/interrupt.h文件中的这个这个枚举,然后在里面添加自己的软中断号就可以了。

**(2)在Linux 系统中，如果想响应中断，需要注册，软中断需要注册吗？**

是的，软中断也需要注册。我们可以通过以下接口，注册我们自己的软中断。

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

功能:注册软中断

参数:

@nr 软中断号

@action 软中断触发的时候，需要执行的软中断处理函数

返回值:

无

**(3)硬件中断，最终是由硬件触发的。软中断是谁来触发的呢？**

软中断，需要软件显示的触发。调用如下函数就可以触发软中断。

```
void raise_softirq(unsigned int nr)
```

功能:触发软中断

参数:

@nr 需要触发的软中断号

返回值:

无

大家可以思考以下，我们应该什么时候触发软中断呢？

嗯，在中断上半部处理函数返回前触发软中断。

(4)软中断处理函数什么时候才会执行呢？

前面我们提到过，Linux 内核会在合适时机执行。这个合适的时机具体指的是什么时候呢？

在Linux 内核中共有三个"时机":

1)一个硬中断返回时，即在irq\_exit中有机会执行软中断。

2)在ksoftirqd内核线程中被执行

3)在哪些显示检查或执行待处理软中断的代码中。如网络系统中的netif\_rx\_ni函数，local\_bh\_enable以及local\_bh\_enable\_ip等函数。

以上这段话摘自<http://blog.chinaunix.net/uid-12216245-id-3956941.html>。

软中断最可能是在一个硬件中断返回时，即在硬件中断上半部处理结束时候。我们简单来看一下Linux 内核在irq\_exit函数中做的事情。

```
/*
 * Exit an interrupt context. Process softirqs if needed and possible:
 */
void irq_exit(void)
{
    account_system_vtime(current);
    trace_hardirq_exit();
    sub_preempt_count(IRQ_EXIT_OFFSET);
    if (!in_interrupt() && local_softirq_pending())
        invoke_softirq();
}
```

探测是否有软中断被触发，如果有就执行

```

    rcu_irq_exit();
#ifdef CONFIG_NO_HZ
    /* Make sure that timer wheel updates are propagated */
    if (idle_cpu(smp_processor_id()) && !in_interrupt() && !need_resched())
        tick_nohz_stop_sched_tick(0);
#endif
    preempt_enable_no_resched();
}

```

貌似看起这段代码有点晕呀！嗯，不急只需要看我用红线框起来的部分。其他的等哪天自己已经变成一只小牛的时候在去看，看完就变成大牛了。初学者切记不要随便读Linux内核代码，不然只能受到打击。**个人觉的正确的学习方法应该是先理解机制，学会使用，然后在去深入了解。**

从上图可以看到执行软中断最终调用了invoke\_softirq()函数。这个函数是一个宏，最终调用了\_\_do\_softirq()函数。我们简单了解一下这个函数内部都做了一些什么？

```

/*
 * We restart softirq processing MAX_SOFTIRQ_RESTART times,
 * and we fall back to softirqd after that.
 *
 * This number has been established via experimentation.
 * The two things to balance is latency against fairness -
 * we want to handle softirqs as soon as possible, but they
 * should not be able to lock up the box.
 */
#define MAX_SOFTIRQ_RESTART 10

asmlinkage void __do_softirq(void)
{
    struct softirq_action *h;
    __u32 pending;
    int max_restart = MAX_SOFTIRQ_RESTART;
    int cpu;

    pending = local_softirq_pending();
    account_system_vtime(current);

    __local_bh_disable((unsigned long)__builtin_return_address(0));

```

```

lockdep_softirq_enter();

cpu = smp_processor_id();

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

local_irq_enable();

h = softirq_vec;

```

```

do {
    if (pending & 1) {
        int prev_count = preempt_count();
        kstat_incr_softirqs_this_cpu(h - softirq_vec);

        trace_softirq_entry(h, softirq_vec);
        h->action(h); 调用软中断处理函数
        trace_softirq_exit(h, softirq_vec);
        if (unlikely(prev_count != preempt_count())) {
            printk(KERN_ERR "huh, entered softirq %td %s %p"
                    "with preempt_count %08x,"
                    " exited with %08x?\n", h - softirq_vec,
                    softirq_to_name[h - softirq_vec],
                    h->action, prev_count, preempt_count());
            preempt_count() = prev_count;
        }

        rcu_bh_qs(cpu);
    }
    h++;
    pending >>= 1;
} while (pending);

```

处理所有已经触发的软中断

```

local_irq_disable();

```

```

local_irq_disable();

pending = local_softirq_pending();
if (pending && --max_restart)
    goto restart;

if (pending)
    wakeup_softirqd();

lockdep_softirq_exit();

account_system_vtime(current);
_local_bh_enable();
}

```

处理结束的时候,判定在处理软中断的过程中是不是又有软中断触发了,如果有则跳到restart标签继续执行。不过也不会一直这样,最多会进行max\_restart次。

经历过max\_restart次循环处理,还是有触发的软中断未处理,则唤醒ksoftirqd内核线程,让它处理软中断。

嗯,代码很多,大家只需要关注我标识的地方就可以了。

从上面代码可以知道,如果软中断触发的过于频繁,Linux 内核最终会唤醒ksoftirqd,让它来处理软中断。正常情况这种事情是不可能发生的,但也不能说绝对不可能。

Linux 内核线程ksoftirqd执行流程:

```

700     preempt_disable();
701     if (!local_softirq_pending()) {
702         preempt_enable_no_resched();
703         schedule();
704         preempt_disable();
705     }
706
707     __set_current_state(TASK_RUNNING);
708
709     while (local_softirq_pending()) {
710         /* Preempt disable stops cpu going offline.
711            If already offline, we'll be on wrong CPU:
712            don't process */
713         if (cpu_is_offline((long)__bind_cpu))
714             goto wait_to_die;
715         do_softirq();
716         preempt_enable_no_resched();

```



```

717     cond_resched();
718     preempt_disable();
719     rcu_note_context_switch((long)__bind_cpu);
720 }
721 preempt_enable();
722 set_current_state(TASK_INTERRUPTIBLE);
723 }
724 set_current_state(TASK_RUNNING);

```

kernel/softirq.c 710,1-4

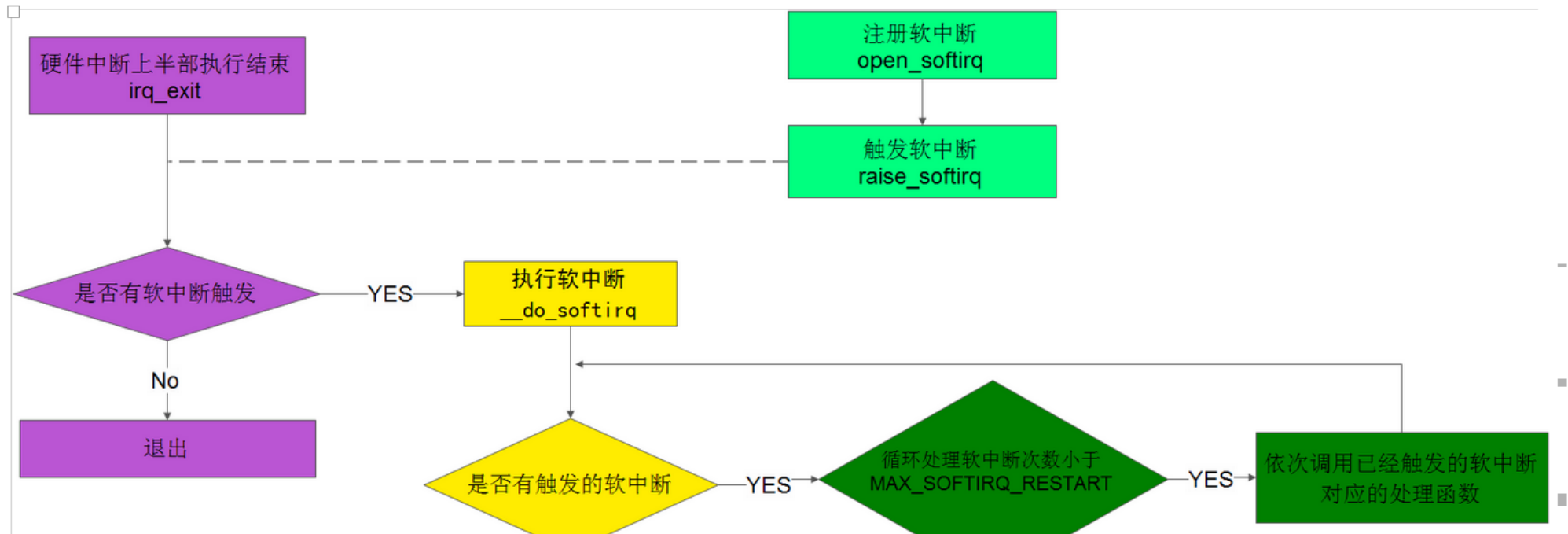
Linux 操作系统为每个CPU都分配了一个ksoftirqd。在ubuntu上搜索ksoftirqd结果如下:

```

cyg@ubuntuk:~/workdir/s5pc100/linux-2.6.35-1$ ps -ef | grep softirq
root      3      2  0 09:48 ?        00:00:00 [ksoftirqd/0]
root     29      2  0 09:48 ?        00:00:00 [ksoftirqd/1]
root     34      2  0 09:48 ?        00:00:00 [ksoftirqd/2]
root     39      2  0 09:48 ?        00:00:00 [ksoftirqd/3]
cyg      3600  3152  0 14:02 pts/0    00:00:00 grep --color=auto softirq

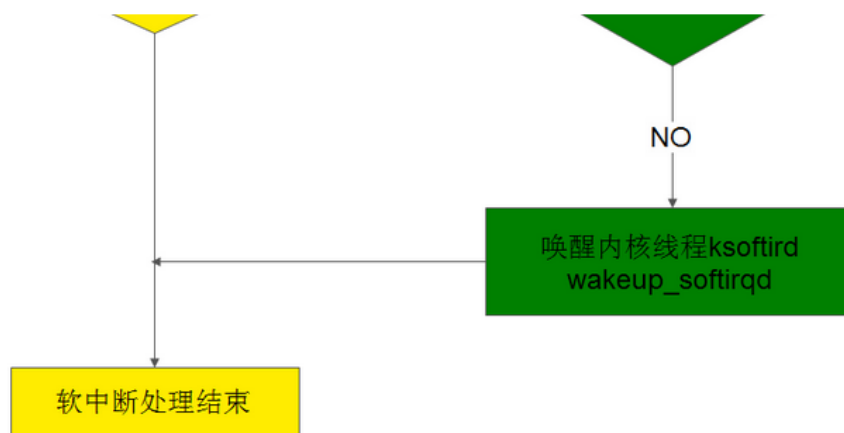
```

好了，到这里大家应该明白了，软中断如何使用，什么时候触发。下面我们简单的总结一下，软中断使用流程。





## CYG制作



分析到这里，要告诉大家一个不幸的消息，Linux 内核并没有导出open\_softirq、raise\_softirq函数的符号，也就是说我们无法以动态加载模块的方式使用软中断。如果大家需要使用软中断，可以修改内核代码，导出这两个函数的符号，然后重新编译内核。

修改的文件：

(1)kernel/softirq.c 修改如下

```
59 char *softirq_to_name[NR_SOFTIRQS] = {
60     "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCK_IOPOLL",
61     "TASKLET", "SCHED", "HRTIMER", "CYG", "RCU"
62 };
63
64
335 void raise_softirq(unsigned int nr)
336 {
337     unsigned long flags;
338
339     local_irq_save(flags);
340     raise_softirq_irqoff(nr);
341     local_irq_restore(flags);
342 }
343 EXPORT_SYMBOL(raise_softirq);
344
345 void open_softirq(int nr, void (*action)(struct softirq_action *))
346 {
347     softirq_vec[nr].action = action;
348 }
349 EXPORT_SYMBOL(open_softirq);
350
```

添加自己的软中断名字

导出我们需要的函数

(2) `include/linux/interrupt.h`修改如下

```
371 enum
372 {
373     HI_SOFTIRQ=0,
374     TIMER_SOFTIRQ,
375     NET_TX_SOFTIRQ,
376     NET_RX_SOFTIRQ,
377     BLOCK_SOFTIRQ,
378     BLOCK_IOPOLL_SOFTIRQ,
379     TASKLET_SOFTIRQ,
380     SCHED_SOFTIRQ,
381     HRTIMER_SOFTIRQ,
382     RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */
383     CYG_SOFTIRQ,
384     NR_SOFTIRQS
385 };
```

添加自己的软中断号

`include/linux/interrupt.h`

(3)重新编译内核:`make zImage`

## 二、tasklet

**tasklet翻译成中文叫做"小任务",它也是基于软中断实现的。**还记的上面提到过Linux 内核支持的10种软中断中的HI\_SOFTIRQ和TASKLET\_SOFTIRQ吗?这两种软中断就是用来实现tasklet的。

你可能会问,既然tasklet是基于软中断实现的,我们为什么不直接使用软中断呢?

**前面我们知道软中断我们是不能直接使用的,所以Linux 内核为了方便我们使用软中断,在软中断之上封装了一层接口,让我们使用起来更简单,也就是tasklet。所以,除非你对性能要求特别高,否则都应该使用tasklet。**

好了,我们已经知道tasklet是基于软中断实现的,原理上大家应该不缺。接下来我们就来看看如何使用tasklet吧!

在Linux 内核中定义了tasklet\_struct 结构体来描述一个tasklet,内容如下:

```
/* 1.If tasklet_schedule() is called, then tasklet is guaranteed
to be executed on some cpu at least once after this.
```

- \* 2.If the tasklet is already scheduled, but its execution is still not started, it will be executed only once.
- \* 3.If this tasklet is already running on another CPU (or schedule is called from tasklet itself), it is rescheduled for later.
- \* 3.Tasklet is strictly(严格) serialized wrt itself, but not wrt another tasklets. If client needs some intertask synchronization, he makes it with spinlocks.\*/\*

```
struct tasklet_struct
{
    //下一个tasklet
    struct tasklet_struct *next;
    //tasklet有两种状态:
    //TASKLET_STATE_SCHED : 表示tasklet已经被调度等待执行
    //TASKLET_STATE_RUN   : 表示tasklet正在被执行,次状态在多处理器系统中有效
    unsigned long state;
    //count值为不为0则禁用tasklet,不允许执行
    //count值为0则允许tasklet被执行
    atomic_t count;
    //tasklet处理函数
    void (*func)(unsigned long);
    //给tasklet处理函数传递的参数
    unsigned long data;
};
```

(1) 初始化一个tasklet

//静态初始化

```
#define DECLARE_TASKLET(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }
```

```
#define DECLARE_TASKLET_DISABLED(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }
```

//动态初始化

```
void tasklet_init(struct tasklet_struct *t,
                 void (*func)(unsigned long), unsigned long data)
{
    t->next = NULL;
    t->state = 0;
```

```
    atomic_set(&t->count, 0);
    t->func = func;
    t->data = data;
}
```

例如:

```
struct tasklet_struct my_tasklet;

tasklet_init(&my_tasklet, func, data);
```

(2) 编写tasklet处理函数

```
void tasklet_handler(unsigned long data)
{
    ....
}
```

(3) 调度tasklet

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}
```

例如:

```
tasklet_schedule(&my_tasklet);
```

在这里需要注意的是一个tasklet被tasklet\_schedule调用过,它的state状态就被设置为TASKLET\_STATE\_SCHED, 接下来等待被执行。tasklet被执行完后, 它的状态才会被清除。也就是说如果重复调用一个tasklet, 它如果没有被执行过, 是不会重复执行的。

实例演示如下:

```
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/module.h>
```

```

#include <linux/interrupt.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("CYG");

void tasklet_handler(unsigned long data)
{
    printk("data = %ld\n", data);
    return;
}

DECLARE_TASKLET(key_tasklet, tasklet_handler, 10);

irqreturn_t key_handler(int irq, void *dev_id)
{
    printk("key1!\n");
    tasklet_schedule(&key_tasklet);
    return IRQ_HANDLED;
}

static int test_init(void)
{
    int ret;

    ret = request_irq(IRQ_EINT(1), key_handler, IRQF_TRIGGER_FALLING, \
        "key1", &key_tasklet);
    if (ret < 0) {
        printk("Failed to request_irq!\n");
        return ret;
    }

    printk("Request IRQ for key1 success!\n");

    return 0;
}

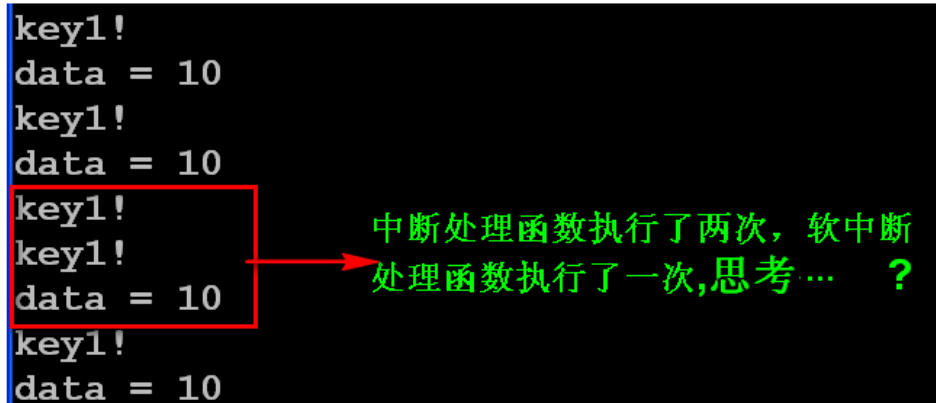
static void test_exit(void)
{

```

```
    printk("Hello, Linux module exit!\n");
    free_irq(IRQ_EINT(1), &key_tasklet);
    return;
}

module_init(test_init);
module_exit(test_exit);
```

测试结果如下:



```
key1!
data = 10
key1!
data = 10
key1!
key1!
data = 10
key1!
data = 10
```

中断处理函数执行了两次, 软中断  
处理函数执行了一次, 思考... ?

### 三、工作队列

工作队列(work queue)是另外一种将工作推后执行的形式, 它和我们前面讨论的所有其他形式都不相同。**工作队列可以把工作推后, 交由一个内核线程去执行—这个下半部分 总是会在进程上下文执行, 但由于是内核线程, 其不能访问用户空间。最重要特点的就是工作队列允许重新调度甚至是睡眠。**

通常, 在工作队列和软中断/tasklet中作出选择非常容易。可使用以下规则:

- (1) 如果推后执行的任务需要睡眠, 那么只能选择工作队列;
- (2) 如果推后执行的任务需要延时指定的时间再触发, 那么使用工作队列, 因为其可以利用timer延时;
- (3) 如果推后执行的任务需要在一个tick之内处理, 则使用软中断或tasklet, 因为其可以抢占普通进程和内核线程;
- (4) 如果推后执行的任务对延迟的时间没有任何要求, 则使用工作队列, 此时通常为无关紧要的任务。

**另外如果你需要用 一个可以重新调度的实体来执行你的下半部处理, 你应该使用工作队列。它是惟一能在进程上下文运行的下半部实现的机制, 也只有它才可以睡眠。这意味着在你需要获得大量的内存时、在你需要获取信号**

量时，在你需要执行阻塞式的I/O操作时，它都会非常有用。

实际上，工作队列的本质就是将工作交给内核线程处理，因此其可以用内核线程替换。但是内核线程的创建和销毁对编程者的要求较高，而工作队列实现了内核线程的封装，不易出错，所以我们也推荐使用工作队列。

接下来，我们就来看看如何使用工作队列。

**核心思想:**我们创建一个工作，加入工作队列，然后唤醒系统默认的工作者线程去执行就可以了。

默认的工作者线程名字叫"events",每个CPU都会有一个和它对应的工作者线程。注意哦，工作者线程与我们的进程参与统一的调度哦。

## 1.创建推后的工作

//对工作的描述结构体

```
struct work_struct
{
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};
```

//工作被工作者线程处理的时候，调用的函数原型

```
typedef void (*work_func_t)(struct work_struct *work);
```

### (1)静态定义

```
DECLARE_WORK(name,void (*work_func_t)(struct work_struct *work));
```



## (2)动态初始化

struct work\_struct work; //定义工作

INIT\_WORK(struct work\_struct \*pwork, void (\*work\_func\_t)(struct work\_struct \*work)); //初始化

## 2.放入工作队列链表,并且唤醒工作者线程

int schedule\_work(struct work\_struct \*pwork);

分析一下, 它的实现过程:

@ kernel/kernel/workqueue.c

```
int schedule_work(struct work_struct *work)
{
    return queue_work(keventd_wq, work);
}
```

keventd\_wq为工作队列

```
int queue_work(struct workqueue_struct *wq, struct work_struct *work)
{
    int ret;
    ret = queue_work_on(get_cpu(), wq, work);
    put_cpu();
    return ret;
}
```

该函数将work工作项提交到当前做该项提交的cpu上的工作队列wq上, 如果这个cpu被标记为die, 那么可以提交到别的cpu上去执行。  
返回0, 表示该项工作已经提交过, 还没执行。非0表示提交成功。

```
int queue_work_on(int cpu, struct workqueue_struct *wq, struct work_struct *work)
{
    int ret = 0;
    if (!test_and_set_bit(WORK_STRUCT_PENDING, work_data_bits(work))) {
        BUG_ON(!list_empty(&work->entry));
    }
}
```

```
    __queue_work(wq_per_cpu(wq, cpu), work); //提交工作，唤醒工作者线程
    ret = 1;
}
return ret;
}
```

总结:如果一个工作没有被执行，后面重复提交的工作将不会被加入工作队列中。

## 四、下半部机制的使用总结

下面对实现中断下半部工作的3种机制进行总结，便于在实际使用中决定使用哪种机制

下半部机制	上下文	复杂度	执行性能	顺序执行保障
软中断	中断	高 (需要自己确保软中断的执行顺序及锁机制)	好 (全部自己实现，便于调优)	没有
tasklet	中断	中 (提供了简单的接口来使用软中断)	中	同类型不能同时执行
工作队列	进程	低 (在进程上下文中运行，与写用户程序差不多)	差	没有 (和进程上下文一样被调度)