链滴

Linux 中断

作者：zhang-ke-wei
原文链接：https://ld246.com/article/1597835267190
来源网站：链滴

<h2 id="一-什么是中断">一、什么是中断</h2>
中断是指计算机运行过程中，出现某些意外情况需主机干预时，机器能自动停止正在运行的程序转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行。中断是频繁使用的功能，中断大的提高了 CPU 的利用率。
<h2 id="二-Linux-中断简介">二、Linux 中断简介</h2>
在 Linux 内核中提供了完善的中断框架，我们只需要申请中断，然后注册中断处理函数即可，使非常方便，不需要一系列复杂的寄存器配置。
<h3 id="1-Linux-中断-API-函数">1.Linux 中断 API 函数</h3>
在裸机中使用中断的流程：
①、使能中断，初始化相应的寄存器。
②、注册中断服务函数，也就是向 irqTable 数组的指定标号处写入中断服务函数
②、中断发生以后进入 IRQ 中断服务函数，在 IRQ 中断服务函数在数组 irqTable 里面查找具体的中处理函数，找到以后执行相应的中断处理函数。
在 Linux 内核中也提供了大量的中断相关的 API 函数，我们来看一下这些跟中断有关的 API 函数：</
>
<h4 id="-1-中断号">(1)中断号</h4>
每个中断都有一个中断号，通过中断号即可区分不同的中断，中断号叫中断线。在 Linux 内核中用一个 int 变量表示中断号。
<h4 id="-2-request-irq-函数">(2)request_irq 函数</h4>
在 Linux 内核中要想使用某个中断是需要申请的， request_irq 函数用于申请中断， request_ir

函数可能会导致睡眠，因此不能在中断上下文或者其他禁止睡眠的代码段中使用 request_irq 函
数。 request_irq 函数会激活(使能)中断，所以不需要我们手动去使能中断， request_irq 函数原型<b
>
如下：
<pre><code class="highlight-chroma"><span class="highlight
cl">int request_irq(unsigned int irq,irq_handler_t handler,
  unsigned l
ng flags,
  const char
name,void *dev)
</code></pre>
irq：要申请中断的中断号。
handler：中断处理函数，当中断发生以后就会执行此中断处理函数。
flags：中断标志，可以在文件 include/linux/interrupt.h 里面查看所有的中断
志。以下是部分中断标志
<table>
<thead>
<tr>
<th>标志</th>
<th>描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQF_SHARED</td>
<td>多个设备共享一个中断线，共享的所有中断都必须指定此标志。
如果使用共享中断的话， r
quest_irq 函数的 dev 参数就是唯一
区分他们的标志。</td>
</tr>
<tr>
<td>IRQF_ONESHOT</td>
<td>单次中断，中断执行一次就结束。</td>
</tr>
<tr>

<td>IRQF_TRIGGER_NONE</td>
<td>无触发。</td>
</tr>
<tr>
<td>IRQF_TRIGGER_RISING</td>
<td>上升沿触发。</td>
</tr>
<tr>
<td>IRQF_TRIGGER_FALLING</td>
<td>下降沿触发。</td>
</tr>
<tr>
<td>IRQF_TRIGGER_HIGH</td>
<td>高电平触发。</td>
</tr>
<tr>
<td>IRQF_TRIGGER_LOW</td>
<td>低电平触发。</td>
</tr>
</tbody>
</table>
各个不同的标志在使用是可以通过“|”运算符组合。
name：中断名字，设置以后可以在/proc/interrupts 文件中看到对应的断名字。
dev： 如果将 flags 设置为 IRQF_SHARED 的话， dev 用来区分不同的中断，般情况下将 dev 设置为设备结构体， dev 会传递给中断处理函数 irq_handler_t 的第二个参数。
返回值： 0 中断申请成功，其他负值 中断申请失败，如果返回-EBUSY 的话表中断已经被申请了。
<h4 id="-3-free-irq-函数">(3)free_irq 函数</h4>
使用中断的时候需要通过 request_irq 函数申请，使用完成以后就要通过 free_irq 函数释放掉相的中断。如果中断不是共享的，那么 free_irq 会删除中断处理函数并且禁止中断。
free_irq 函数原型如下所示：
<pre><code class="highlight-chroma">void free_irq(unsigned int irq, void *dev)
</code></pre>
irq： 要释放的中断。
dev：如果中断设置为共享(IRQF_SHARED)的话，此参数用来区分具体的中断共享中断只有在释放最后中断处理函数的时候才会被禁止掉。
返回值：无。
<h4 id="-4-中断处理函数">(4)中断处理函数</h4>
使用 request_irq 函数申请中断的时候需要设置中断处理函数，中断处理函数格式如下所示：</
>
<pre><code class="highlight-chroma">irqreturn_t (*irq_handler_t) (int, void *)
</code></pre>
第一个参数是要中断处理函数要相应的中断号。第二个参数是一个指向 void 的指针，也就是个用指针，需要与 request_irq 函数的 dev 参数保持一致。用于区分共享中断的不同设备，dev 也可以向设备数据结构。中断处理函数的返回值为 irqreturn_t 类型， irqreturn_t 类型定义如下所示：
<pre><code class="highlight-chroma">enum irqreturn {
    IRQ_NONE = (0 << 0), /* 不是本设备中断 */
    IRQ_HANDLED = (1 << 0), /* 本设备中断 */

    IRQ_WAKE_THEAD = (1 << 1), /* 处理程序请求线程唤醒 */
};
typedef enum irqrturn irqreturn_t;
</code></pre>
irqreturn_t 是个枚举类型，一共有三种返回值。一般中断服务函数返回值使用如下形式：
<pre><code class="highlight-chroma">return IRQ_RETVAL(IRQ_HANDLED)
</code></pre>
<h4 id="-5-中断使能与禁止函数">(5)中断使能与禁止函数</h4>
常用的中断使用和禁止函数如下所示：
<pre><code class="highlight-chroma">void enable_irq(unsigned int irq)
void disable_irq(usigned int irq)
</code></pre>
enable_irq 和 disable_irq 用于使能和禁止指定的中断， irq 就是要禁止的中断号。 disable_irq 函数要等到当前正在执行的中断处理函数执行完才返回，因此使用者需要保证不会产生新的中断，并确保所有已经开始执行的中断处理程序已经全部退出。在这种情况下，可以使用另外一个中断禁止函：
<pre><code class="highlight-chroma">void disable_irq_nosync(unsigned int irq)
</code></pre>
disable_irq_nosync 函数调用以后立即返回，不会等待当前中断处理程序执行完毕。
上面三个函数都是使能或者禁止某一个中断，有时候我们需要关闭当前处理器的整个中断系统.</>
<pre><code class="highlight-chroma">local_irq_enable()
local_irq_disable()
</code></pre>
local_irq_enable 用于使能当前处理器中断系统， local_irq_disable 用于禁止当前处理器中断统。
<pre><code class="highlight-chroma">local_irq_save(flags)
local_irq_restore(flgs)
</code></pre>
这两个函数是一对， local_irq_save 函数用于禁止中断，并且将中断状态保存在 flags 中。local_rq_restore 用于恢复中断，将中断到 flags 状态。
<h3 id="2-上半部与下半部">2.上半部与下半部</h3>
我们总是期望中断执行得越快越好，但是实际某些中断任务必须花费相对较长的时间才能够执行毕，那么怎么解决这个问题呢？将中断分为上半部和下半部。
上半部：上半部就是中断处理函数，那些处理过程比较快，不会占用很长时间的处理就可以放在半部完成。
下半部：如果中断处理过程比较耗时，那么就将这些比较耗时的代码提出来，交给下半部去执行，这中断处理函数就会快进快出。
下半部的目的在于讲并不是非常紧急的工作推后执行，非紧迫可中断。至于哪些代码属于上半部哪些代码属于下半部并没有明确的规定，一切根据实际使用情况去判断，这个就很考验驱动编写人员功底了。
参考点：
①、如果要处理的内容不希望被其他中断打断，那么可以放到上半部。
②、如果要处理的任务对时间敏感，可以放到上半部。
③、如果要处理的任务与硬件有关，可以放到上半部。

④、除了上述三点以外的其他任务，优先考虑放到下半部。
中断下半部机制：
<h4 id="-1-软中断">(1)软中断</h4>
Linux 内核使用结构体 softirq_action 表示软中断， softirq_action 结构体定义在文件 include/linux/interrupt.h 中
<pre><code class="highlight-chroma">struct softirq_action
{
    void (*action)(struct softirq_action *);
};
</code></pre>
在 kernel/softirq.c 文件中一共定义了 10 个软中断：
<pre><code class="highlight-chroma">
static struct softirq_action softirq_vec[NR_SOFTIRQS];
</code></pre>
NR_SOFTIRQS 是枚举类型，定义在文件 include/linux/interrupt.h 中，定义如下：
<pre><code class="highlight-chroma">

enum
{
    HI_SOFTIRQ=0, /* 高优先级软中断 */
    TIMER_SOFTIRQ /* 定时器软中断 */
    NET_TX_SOFTIRQ, /* 网络数据发送软中断 */
    NET_RX_SOFTIRQ, /* 网络数据接收软中断 */
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ, /* tasklet 软中断 */
    SCHED_SOFTIRQ, /* 调度软中断 */
    HRTIMER_SOFTIRQ, /* 高精度定时器软中断 */
    RCU_SOFTIRQ, /* RCU 软中断 */
    NR_SOFTIRQS
};
</code></pre>
softirq_action 结构体中的 action 成员变量就是软中断的服务函数，数组 softirq_vec 是个全局组，因此所有的 CPU(对于 SMP 系统而言)都可以访问到，每个 CPU 都有自己的触发和控制机制，并只执行自己所触发的软中断。但是各个 CPU 所执行的软中断服务函数确是相同的，都是数组 softirq_vec 中定义的 action 函数。要使用软中断，必须先使用 open_softirq 函数注册对应的软中断处理函， open_softirq 函数原型如下：
<pre><code class="highlight-chroma">
void open_softirq(int nr, void (*action)(struct softirq_action *))
</code></pre>
nr：要开启的软中断，在示例代码 51.1.2.3 中选择一个。
action：软中断对应的处理函数。
返回值： 没有返回值。

注册好软中断以后需要通过 raise_softirq 函数触发， raise_softirq 函数原型如下：
<pre><code class="highlight-chroma">void raise_softirq(unsigned int nr)
</code></pre>
nr：要触发的软中断，十选一
返回值： 没有返回值。
软中断必须在编译的时候静态注册！ Linux 内核使用 softirq_init 函数初始化软中断，
softirq_init 函数定义在 kernel/softirq.c 文件里面，函数内容如下：
<pre><code class="highlight-chroma">void __init softirq_init(void)
{
    int cpu;

    for_each_possible_cpu(cpu) {
        per_cpu(tasklet_vec, cpu).tail =
        &per_cu(tasklet_vec, cpu).head;
        per_cpu(tasklet_hi_vec, cpu).tail =
        &per_cu(tasklet_hi_vec, cpu).head;
    }

    open_softirq(TASKLET_SOFTIRQ, tasklet_action);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}
</code></pre>
softirq_init 函数默认会打开 TASKLET_SOFTIRQ 和 HI_SOFTIRQ。
<h4 id="-2-tasklet">(2)tasklet</h4>
tasklet 是利用软中断来实现的另外一种下半部机制。软中断和 tasklet 有密切的关系，tasklet 在软中断之上实现。
软中断的分配是静态的(即在编译时定义)，而 tasklet 的分配和初始化可以在运行时进行(例如:安一个内核模块时)。软中断(即便是同一种类型的软中断)可以并发地运行在多个 CPU 上。因此，软中是可重入函数而且必须明确地使用自旋锁保护其数据结构。tasklet 不必担心这些问题，因为内核对 tasklet 的执行进行了更加严格的控制。相同类型的 tasklet 总是被串行地执行，换句话说就是:不能在两个 CPU 上同时运行相同类型的 tasklet。但是，类型不同的 tasklet 可以在几个 CPU 上并发执行。taskle 的串行化使 tasklet 函数不必是可重入的，因此简化了设备驱动程序开发者的工作。在软中断和 tasklet 之间，建议使用 tasklet。
tasklet_struct 结构体
<pre><code class="highlight-chroma">struct tasklet_struct
{
    struct tasklet_struct *next; /* 下一个 tasklet */
    unsigned long state; /* tasklet 状态 */
    atomic_t count; * 计数器，记录对 tasklet 的引用数 */
    void (*func)(unsigned long); /* tasklet 执行的函数 */

    unsigned long ata; /* 函数 func 的参数 */
};
</code></pre>
func 函数就是 tasklet 要执行的处理函数，用户定义函数内容，相当于中断处理函数。如果要使 tasklet，必须先定义一个 tasklet，然后使用 tasklet_init 函数初始化 tasklet，taskled_init 函数原如下：
<pre><code class="highlight-chroma">void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data);
</code></pre>
t：要初始化的 tasklet
func： tasklet 的处理函数。
data： 要传递给 func 函数的参数
返回值： 没有返回值。
也可以使用宏 DECLARE_TASKLET 来一次性完成 tasklet 的定义和初始化，DECLARE TASKLET 定义在 include/linux/interrupt.h 文件中，定义如下:
<pre><code class="highlight-chroma">DECLARE_TASKLET(name, func, data)
</code></pre>
其中 name 为要定义的 tasklet 名字，这个名字就是一个 tasklet_struct 类型的时候变量， func 就是 tasklet 的处理函数， data 是传递给 func 函数的参数。
在上半部，也就是中断处理函数中调用 tasklet_schedule 函数就能使 tasklet 在合适的时间运行 tasklet_schedule 函数原型如下：
<pre><code class="highlight-chroma">void tasklet_schedule(struct tasklet_struct *t)

</code></pre>
函数参数和返回值含义如下：
t：要调度的 tasklet，也就是 DECLARE_TASKLET 宏里面的 name。
返回值： 没有返回值。
exampel:
<pre><code class="highlight-chroma">/* 定义 taselet */
struct tasklet_struct testtasklet;

/* tasklet 处理函数 */
void testtasklet_func(unsigned long data)
{
    /* tasklet 具体处理内容 */
}

/* 中断处理函数 */
irqreturn_t test_handler(int irq, void *dev_id)
{
    ......
    /* 调度 tasklet *

    tasklet_schedul(&testtasklet);

    ......
}

/* 驱动入口函数 */
static int __init xxx_init(void)
{
    ......
    /* 初始化 tasklet */
    tasklet_init(&testtasklet, testtasklet_func, data);
    /* 注册中断处函数 */
    request_irq(xxx_irq, test_handler, 0, "xxx", &xxx_dev);
    ......
}
</code></pre>
<h4 id="-3-工作队列">(3)工作队列</h4>
工作队列是另外一种下半部执行方式，工作队列在进程上下文执行，工作队列将要推后的工作交一个内核线程去执行，因为工作队列工作在进程上下文，因此工作队列允许睡眠或重新调度。因此如你要推后的工作可以睡眠那么就可以选择工作队列，否则选择软中断或 tasklet。
Linux 内核使用 work_struct 结构体表示一个工作:
<pre><code class="highlight-chroma">struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t fun; /* 工作队列处理函数 */
};
</code></pre>
这些工作组织成工作队列，工作队列使用 workqueue_struct 结构体表示:
<pre><code class="highlight-chroma">struct workqueue_struct {
    struct list_head pwqs;
    struct list_head list;
    struct mutex mtex;
    int work_color;
    int flush_color;
    atomic_t nr_pws_to_flush;
    struct wq_flusher *first_flusher;
    struct list_head flusher_queue;
    struct list_head flusher_overflow;
    struct list_head

maydays;
    struct worker *r
scuer;
    int nr_drainers;
    int saved_max_
ctive;
    struct workque
e_attrs *unbound_attrs;
    struct pool_wor
queue *dfl_pwq;
    char name[WQ
NAME_LEN];
    struct rcu_head
cu;
    unsigned int fla
s ___cacheline_aligned;
    struct pool_wor
queue __percpu *cpu_pwqs;
    struct pool_wor
queue __rcu *numa_pwq_tbl[];
};
</code></pre>
Linux 内核使用工作者线程(worker thread)来处理工作队列中的各个工作， Linux 内核使用 wor
er 结构体表示工作者线程， worker 结构体内容如下：
<pre><code class="highlight-chroma"><span class="highlight
cl">struct worker {
    union {
        struct list_he
d entry;
        struct hlist_n
de hentry;
    };
    struct work_str
ct *current_work;
    work_func_t cur
ent_func;
    struct pool_wor
queue *current_pwq;
    bool desc_valid;
    struct list_head
cheduled;
    struct task_struc
 *task;
    struct worker_p
ol *pool;
    struct list_head
ode;
    unsigned long l
st_active;
    unsigned int fla
s;
    int id;
    char desc[WOR
ER_DESC_LEN];

    struct workque
e_struct *rescue_wq;
};
</code></pre>
每个 worker 都有一个工作队列，工作者线程处理自己工作队列中的所有工作。在驱动开发中，
们只需要定义工作(work_struct)即可，关于工作队列和工作者线程我们基本不用去管。简单创建工作
简单，直接定义一个 work_struct 结构体变量即可，然后使用 INIT_WORK 宏来初始化工作， INIT_
ORK 宏定义如下：
<pre><code class="highlight-chroma"><span class="highlight
cl">#define INIT_WORK(_work, _func)
</code></pre>
work 表示要初始化的工作， _func 是工作对应的处理函数。也可以使用 DECLARE_WORK 宏
次性完成工作的创建和初始化，宏定义如下：
<pre><code class="highlight-chroma"><span class="highlight
cl">#define DECLARE_WORK(n, f)
</code></pre>
n 表示定义的工作(work_struct)，
f 表示工作对应的处理函数。和 tasklet 一样，工作也是需要调度才能运行
，工作的调度函数为 schedule_work，函数原型如下所示：
<pre><code class="highlight-chroma"><span class="highlight
cl">bool schedule_work(struct work_struct *work)
</code></pre>
work： 要调度的工作。
返回值： 0 成功，其他值 失败。
example:
<pre><code class="highlight-chroma"><span class="highlight
cl">/* 定义工作(work) */
struct work_struct
estwork;

/* work 处理函数 *

void testwork_fun
_t(struct work_struct *work);
{
    /* work 具体处
内容 */
}

/* 中断处理函数 */
irqreturn_t test_h
ndler(int irq, void *dev_id)
{
    ......
    /* 调度 work */
    schedule_work
&testwork);
    ......
}

/* 驱动入口函数 */
static int __init xxx
_init(void)
{

    .....
    /* 初始化 work *

    INIT_WORK(&
mp;testwork, testwork_func_t);
    /* 注册中断处
函数 */
    request_irq(xxx_
rq, test_handler, 0, "xxx", &xxx_dev);
    .....
}
</code></pre>
<h3 id="3-获取中断号">3 获取中断号</h3>
编写驱动的时候需要用到中断号，我们用到中断号，中断信息已经写到了设备树里面，因此可以
过 irq_of_parse_and_map 函数从 interupts 属性中提取到对应的设备号，函数原型如下：
<pre><code class="highlight-chroma"><span class="highlight
cl">unsigned int irq_of_parse_and_map(struct device_node *dev, int index)
</code></pre>
dev： 设备节点。
index：索引号， interrupts 属性可能包含多条中断信息，通过 index 指
要获取的信息。
返回值：中断号。
如果使用 GPIO 的话，可以使用 gpio_to_irq 函数来获取 gpio 对应的中断号，函数原型如下：
<pre><code class="highlight-chroma"><span class="highlight
cl">int gpio_to_irq(unsigned int gpio)
</code></pre>
gpio： 要获取的 GPIO 编号。
返回值： GPIO 对应的中断号。