18. Linux 系统对中断的处理

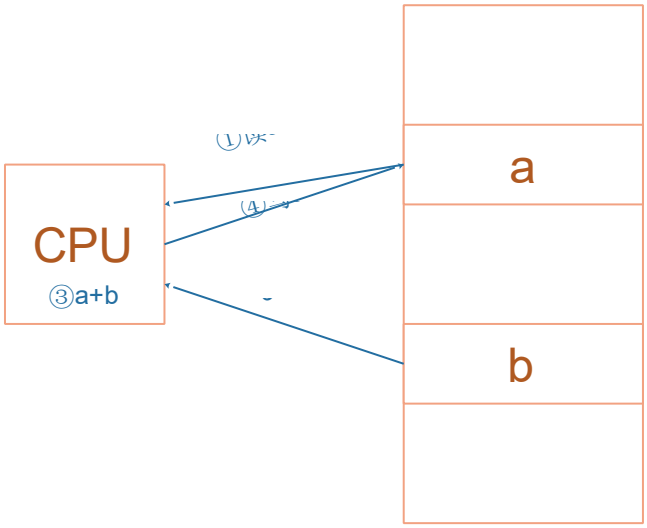
18.1 进程、线程、中断的核心：栈

中断中断，中断谁？
中断当前正在运行的进程、线程。
进程、线程是什么？内核如何切换进程、线程、中断？
要理解这些概念，必须理解栈的作用。

18.1.1 ARM 处理器程序运行的过程

ARM 芯片属于精简指令集计算机(RISC：Reduced Instruction Set Computing)，它所用的指令比较简单，有如下特点：
① 对内存只有读、写指令
② 对于数据的运算是在 CPU 内部实现
③ 使用 RISC 指令的 CPU 复杂度小一点，易于设计

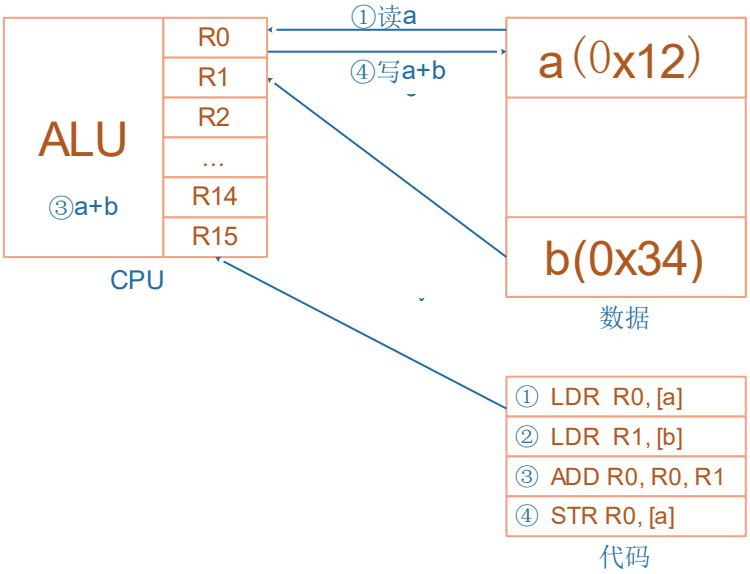
比如对于 a=a+b 这样的算式，需要经过下面 4 个步骤才可以实现：



细看这几个步骤，有些疑问：
① 读 a，那么 a 的值读出来后保存在 CPU 里面哪里？
② 读 b，那么 b 的值读出来后保存在 CPU 里面哪里？
③ a+b 的结果又保存在哪里？

我们需要深入 ARM 处理器的内部。简单概括如下，我们先忽略各种 CPU 模式(系统模式、用户模式等等)。
注意：如果想入理解 ARM 处理器架构，应该从裸机开始学习。我们即将写好近 30 个裸机程序的文档，估计还 3 月底发布。

注意：为了加快学习速度，建议先不看裸机。



CPU 运行时，先去取得指令，再执行指令：

① 把内存 a 的值读入 CPU 寄存器 R0
② 把内存 b 的值读入 CPU 寄存器 R1
③ 把 R0、R1 累加，存入 R0
④ 把 R0 的值写入内存 a

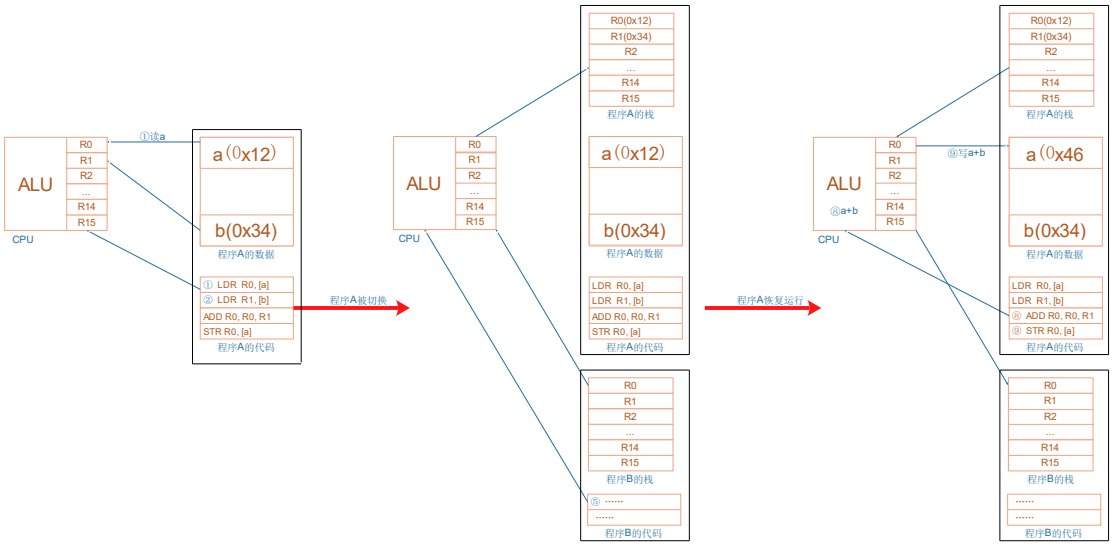
18.1.2 程序被中断时，怎么保存现场

从上图可知，CPU 内部的寄存器很重要，如果要暂停一个程序，中断一个程序，就需要把这些寄存器的值保存下来：这就称为保存现场。

保存在哪里？内存，这块内存就称之为栈。

程序要继续执行，就先从栈中恢复那些 CPU 内部寄存器的值。

这个场景并不局限于中断，下图可以概括程序 A、B 的切换过程，其他情况是类似的：



a. 函数调用：

在函数 A 里调用函数 B，实际就是中断函数 A 的执行。

那么需要把函数 A 调用 B 之前瞬间的 CPU 寄存器的值，保存到栈里；
再去执行函数 B；
函数 B 返回之后，就从栈中恢复函数 A 对应的 CPU 寄存器值，继续执行。

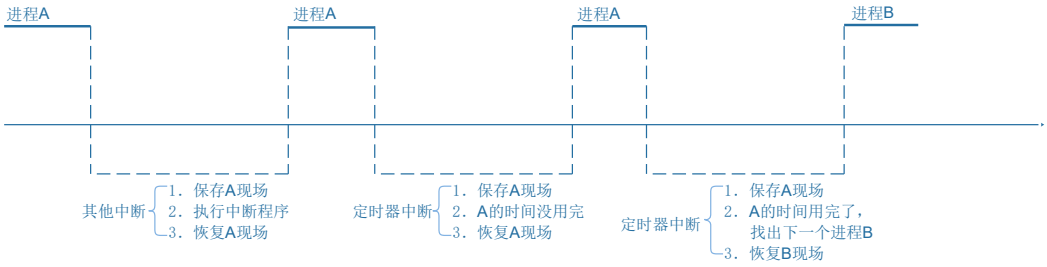
b. 中断处理

进程 A 正在执行，这时候发生了中断。
CPU 强制跳到中断异常向量地址去执行，
这时就需要保存进程 A 被中断瞬间的 CPU 寄存器值，
可以保存在进程 A 的内核态栈，也可以保存在进程 A 的内核结构体中。
中断处理完毕，要继续运行进程 A 之前，恢复这些值。

c. 进程切换

在所谓的多任务操作系统中，我们以为多个程序是同时运行的。
如果我们能感知微秒、纳秒级的事件，可以发现操作系统时让这些程序依次执行一小段时间，进程 A 的时间用完了，就切换到进程 B。
怎么切换？
切换过程是发生在内核态里的，跟中断的处理类似。
进程 A 的被切换瞬间的 CPU 寄存器值保存在某个地方；
恢复进程 B 之前保存的 CPU 寄存器值，这样就可以运行进程 B 了。

所以，在中断处理的过程中，伴存着进程的保存现场、恢复现场。
进程的调度也是使用栈来保存、恢复现场：



18.1.3 进程、线程的概念

假设我们写一个音乐播放器，在播放音乐的同时会根据按键选择下一首歌。把事情简化为 2 件事：发送音频数据、读取按键。那可以这样写程序：

```
int main(int argc, char **argv)
{
    int key;

    while (1)
    {
        key = read_key();

        if (key != -1)
        {
```

```
                switch (key)
                {
                    case NEXT:
                        select_next_music(); //  在 GUI 选中下一首歌
                        break;
                }
            }
            else
            {
                send_music();
            }
        }

        return 0;
}
```

这个程序只有一条主线，读按键、播放音乐都是顺序执行。

无论按键是否被按下，read_key 函数必须马上返回，否则会使得后续的 send_music 受到阻滞导致音乐播放不流畅。

读取按键、播放音乐能否分为两个程序进行？可以，但是开销太大：读按键的程序，要把按键通知播放音乐的程序，进程间通信的效率没那么高。

这时可以用多线程之编程，读取按键是一个线程，播放音乐是另一个线程，它们之间可以通过全局变量传递数据，示意代码如下：

```
int g_key;
void key_thread_fn()
{
    while (1)
    {
        g_key = read_key();
        if (g_key != -1)
        {
            switch (g_key)
            {
                case NEXT:
                    select_next_music(); //  在 GUI 选中下一首歌
                    break;
            }
        }
    }
}

void music_fn()
{
    while (1)
    {
```

```
            if (g_key == STOP)
                stop_music();
            else
            {
                send_music();
            }
        }

}

int main(int argc, char **argv)
{
        int key;

        create_thread(key_thread_fn);
        create_thread(music_fn);
        while (1)
        {
            sleep(10);
        }
        return 0;
}
```

这样，按键的读取及 GUI 显示、音乐的播放，可以分开来，不必混杂在一起。

按键线程可以使用阻塞方式读取按键，无按键时是休眠的，这可以节省 CPU 资源。

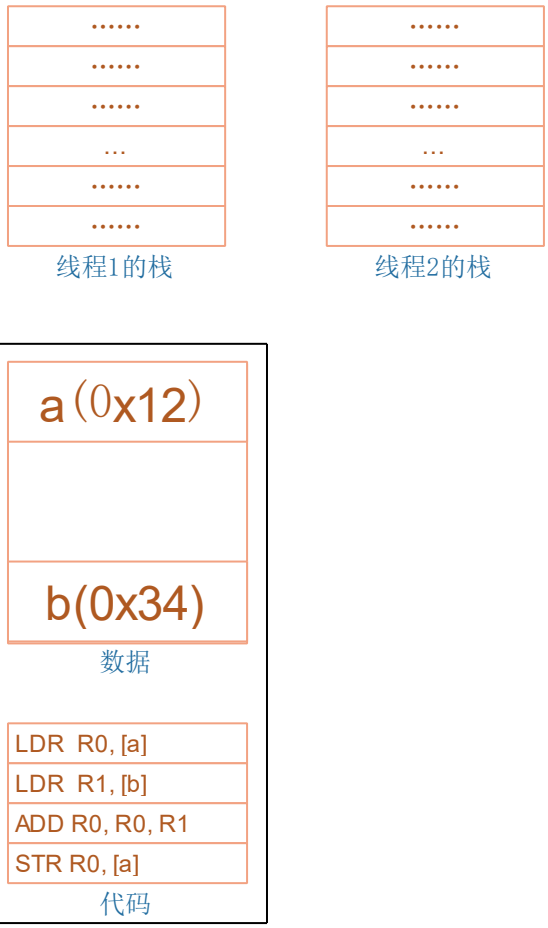
音乐线程专注于音乐的播放和控制，不用理会按键的具体读取工作。

并且这 2 个线程通过全局变量 g_key 传递数据，高效而简单。

在 Linux 中：资源分配的单位是进程，调度的单位是线程。

也就是说，在一个进程里，可能有多个线程，这些线程共用打开的文件句柄、全局变量等等。

而这些线程，之间是互相独立的，“同时运行”，也就是说：每一个线程，都有自己的栈。如下图示：


.....
.....
.....
...
.....
.....
线程1的栈
.....
.....
.....
...
.....
.....
线程2的栈
a(0x12)
b(0x34)
数据
LDR R0, [a]
LDR R1, [b]
ADD R0, R0, R1
STR R0, [a]
代码
本进程里所有线程共享

18.2 Linux 系统对中断处理的演进

从 2005 年我接触 Linux 到现在 15 年了，Linux 中断系统的变化并不大。比较重要的就是引入了 threaded irq：使用内核线程来处理中断。

Linux 系统中有硬件中断，也有软件中断。

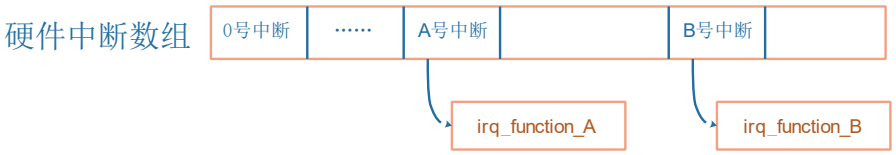
对硬件中断的处理有 2 个原则：不能嵌套，越快越好。

参考资料：https://blog.csdn.net/myarrow/article/details/9287169

18.2.1 Linux 对中断的扩展：硬件中断、软件中断

Linux 系统把中断的意义扩展了，对于按键中断等硬件产生的中断，称之为“硬件中断”(hard irq)。每个硬件中断都有对应的处理函数，比如按键中断、网卡中断的处理函数肯定不一样。

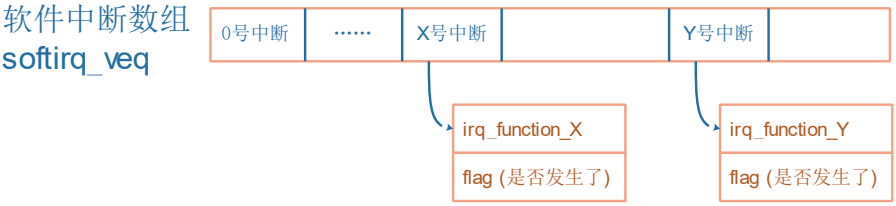
为方便理解，你可以先认为对硬件中断的处理是用数组来实现的，数组里存放的是函数指针：



注意：上图是简化的，Linux 中这个数组复杂多了。

当发生 A 中断时，对应的 irq_function_A 函数被调用。硬件导致该函数被调用。

相对的，还可以人为地制造中断：软件中断(soft irq)，如下图所示：



注意：上图是简化的，Linux 中这个数组复杂多了。

问题来了：

a. 软件中断何时生产？

由软件决定，对于 X 号软件中断，只需要把它的 flag 设置为 1 就表示发生了该中断。

b. 软件中断何时处理？

软件中断嘛，并不是那么十万火急，有空再处理它好了。

什么时候有空？不能让它一直等吧？

Linux 系统中，各种硬件中断频繁发生，至少定时器中断每 10ms 发生一次，那取个巧？

在处理完硬件中断后，再去处理软件中断？就这么办！

有哪些软件中断?

查内核源码 include/linux/interrupt.h

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    IRQ_POLL_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
};
```

怎么触发软件中断?最核心的函数是 raise_softirq,简单地理解就是设置 softirq_veq[nr]的标记位:

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

怎么设置软件中断的处理函数:

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

后面讲到的中断下半部 tasklet 就是使用软件中断实现的。

18.2.2 中断处理原则 1:不能嵌套

官方资料:中断处理不能嵌套

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e58aa3d2d0cc

中断处理函数需要调用 C 函数,这就需要用到栈。

中断 A 正在处理的过程中,假设又发生了中断 B,那么在栈里要保存 A 的现场,然后处理 B。

在处理 B 的过程中又发生了中断 C,那么在栈里要保存 B 的现场,然后处理 C。

如果中断嵌套突然暴发,那么栈将越来越大,栈终将耗尽。

所以,为了防止这种情况发生,也是为了简单化中断的处理,在 Linux 系统上中断无法嵌套:即当前中断 A 没处理完之前,不会响应另一个中断 B(即使它的优先级更高)。

18.2.3 中断处理原则 2:越快越好

妈妈在家中照顾小孩时,门铃响起,她开门取快递:这就是中断的处理。她取个快递敢花上半天吗?不怕小孩出意外吗?

同理,在 Linux 系统中,中断的处理也是越快越好。

在单芯片系统中,假设中断处理很慢,那应用程序在这段时间内就无法执行:系统显得

很迟顿。

在 SMP 系统中，假设中断处理很慢，那么正在处理这个中断的 CPU 上的其他线程也无法执行。

在中断的处理过程中，该 CPU 是不能进行进程调度的，所以中断的处理要越快越好，尽早让其他中断能被处理——进程调度靠定时器中断来实现。

在 Linux 系统中使用中断是挺简单的，为某个中断 irq 注册中断处理函数 handler，可以使用 request_irq 函数：

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
        const char *name, void *dev)
```

在 handler 函数中，代码尽可能高效。

但是，处理某个中断要做的事情就是很多，没办法加快。比如对于按键中断，我们需要等待几十毫秒消除机械抖动。难道要在 handler 中等待吗？对于计算机来说，这可是一个段很长的时间。

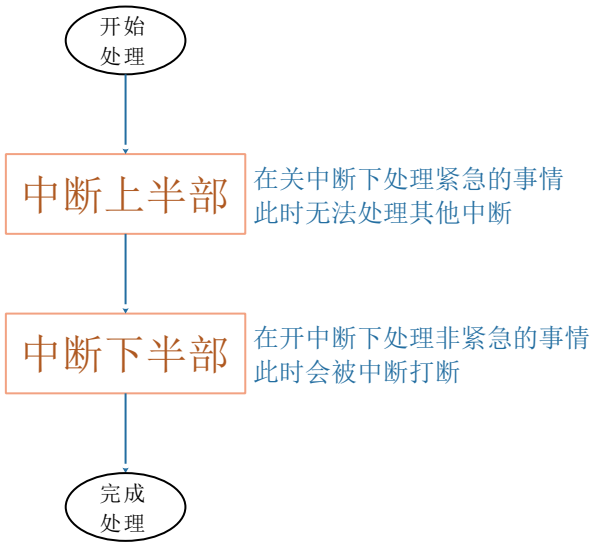
怎么办？

18.2.4 要处理的事情实在太多，拆分为：上半部、下半部

当一个中断要耗费很多时间来处理时，它的坏处是：在这段时间内，其他中断无法被处理。换句话说，在这段时间内，系统是关中断的。

如果某个中断就是要做那么多事，我们能不能把它拆分成两部分：紧急的、不紧急的？

在 handler 函数里只做紧急的事，然后就重新开中断，让系统得以正常运行；那些不紧急的事，以后再处理，处理时是开中断的。



中断下半部的实现有很多种方法，讲 2 种主要的：tasklet(小任务)、work queue(工作队列)。

18.2.5 下半部要做的事情耗时不是太长：tasklet

假设我们把中断分为上半部、下半部。发生中断时，上半部下半部的代码何时、如何被调用？

当下半部比较耗时但是能忍受，并且它的处理比较简单时，可以用 tasklet 来处理下半部。tasklet 是使用软件中断来实现。

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    IRQ_POLL_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
};
```

tasklet软件中断，
用来处理中断下半部

写字太多，不如贴代码，代码一目了然：

```
__irq_svc () at arch/arm/kernel/entry-armv.S:219
    gic_handle_irq at drivers/irqchip/irq-gic.c:364
        handle_domain_irq at ./include/linux/irqdesc.h:168
            __handle_domain_irq at kernel/irq/irqdesc.c:627
                irq_enter();
                    __irq_enter()
                        preempt_count_add(HARDIRQ_OFFSET);  // preempt_count++,表示开始处理中断
                generic_handle_irq(irq); // 会调用reqeust_irq注册的中断函数：上半部
                irq_exit();
                    preempt_count_sub(HARDIRQ_OFFSET);      // preempt_count--, 表示硬件中断处理完毕
                    if (!in_interrupt() && local_softirq_pending()) // in_interrupt: preempt_count > 0
                        invoke_softirq();
                            __do_softirq();
                                __local_bh_disable_ip(_RET_IP_, SOFTIRQ_OFFSET);  // preempt_count++,开始软件中断
                                local_irq_enable();         // 开中断
                                    // 执行softirq_vec数组上各类软中断：下半部
                                local_irq_disable();        // 关中断，马上就会再开中断
                                __local_bh_enable(SOFTIRQ_OFFSET);              // preempt_count--,完成软件中断
```

进出硬件中断

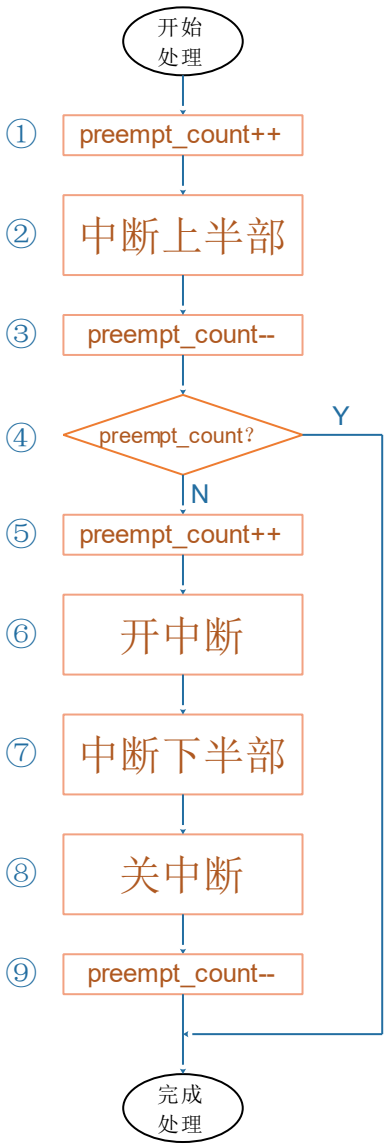
用preempt_count避免软中断多次执行

处理软中断时开中断

上半部

下半部

使用流程图简化一下：



假设硬件中断 A 的上半部函数为 irq_top_half_A，下半部为 irq_bottom_half_A。

使用情景化的分析，才能理解上述代码的精华。

a. 硬件中断 A 处理过程中，没有其他中断发生：

一开始，preempt_count = 0；

上述流程图①~⑨依次执行，上半部、下半部的代码各执行一次。

b. 硬件中断 A 处理过程中，又再次发生了中断 A：

一开始，preempt_count = 0；

执行到第⑥时，一开中断后，中断 A 又再次使得 CPU 跳到中断向量表。

注意：这时 preempt_count 等于 1，并且中断下半部的代码并未执行。

CPU 又从①开始再次执行中断 A 的上半部代码：

在第①步 preempt_count 等于 2；

在第③步 preempt_count 等于 1；

在第④步发现 preempt_count 等于 1，所以直接结束当前第 2 次中断的处理；

注意：重点来了，第 2 次中断发生后，打断了第一次中断的第⑦步处理。当第 2 次中断处理完毕，CPU 会继续去执行第⑦步。

可以看到，发生 2 次硬件中断 A 时，它的上半部代码执行了 2 次，但是下半部代码只执行了一次。

所以，同一个中断的上半部、下半部，在执行时是多对一的关系。

c. 硬件中断 A 处理过程中，又再次发生了中断 B：

一开始，preempt_count = 0；

执行到第⑥时，一开中断后，中断 B 又再次使得 CPU 跳到中断向量表。

注意：这时 preempt_count 等于 1，并且中断 A 下半部的代码并未执行。

CPU 又从①开始再次执行中断 B 的上半部代码：

在第①步 preempt_count 等于 2；

在第③步 preempt_count 等于 1；

在第④步发现 preempt_count 等于 1，所以直接结束当前第 2 次中断的处理；

注意：重点来了，第 2 次中断发生后，打断了第一次中断 A 的第⑦步处理。当第 2 次中断 B 处理完毕，CPU 会继续去执行第⑦步。

在第⑦步里，它会去执行中断 A 的下半部，也会去执行中断 B 的下半部。

所以，多个中断的下半部，是汇集在一起处理的。

总结：

a. 中断的处理可以分为上半部，下半部

b. 中断上半部，用来处理紧急的事，它是在关中断的状态下执行的

c. 中断下半部，用来处理耗时的、不那么紧急的事，它是在开中断的状态下执行的

d. 中断下半部执行时，有可能会被多次打断，有可能会再次发生同一个中断

e. 中断上半部执行完后，触发中断下半部的处理

f. 中断上半部、下半部的执行过程中，不能休眠：中断休眠的话，以后谁来调度进程啊？

18.2.6 下半部要做的事情太多并且很复杂：工作队列

在中断下半部的执行过程中，虽然是开中断的，期间可以处理各类中断。但是毕竟整个中断的处理还没走完，这期间 APP 是无法执行的。

假设下半部要执行 1、2 分钟，在这 1、2 分钟里 APP 都是无法响应的。

这谁受得了？

所以，如果中断要做的事情实在太耗时，那就不能用软件中断来做，而应该用内核线程来做：在中断上半部唤醒内核线程。内核线程和 APP 都一样竞争执行，APP 有机会执行，系统不会卡顿。

这个内核线程是系统帮我们创建的，一般是 kworker 线程，内核中有很多这样的线程：

```
book@book-virtual-machine:~$ ps -A |grep kworker
    4 ?        00:00:00 kworker/0:0H
   18 ?        00:00:00 kworker/1:0H
   24 ?        00:00:00 kworker/2:0H
   30 ?        00:00:00 kworker/3:0H
   36 ?        00:00:00 kworker/4:0H
   42 ?        00:00:00 kworker/5:0H
```

kworker 线程要去“工作队列”(work queue)上取出一个一个“工作”(work)，来执行它里面的函数。

那我们怎么使用 work、work queue 呢？

a. 创建 work：

你得先写出一个函数，然后用这个函数填充一个 work 结构体。比如：

```
static DECLARE_WORK(aer_recover_work, aer_recover_work_func);
```

work结构体　　函数

b. 要执行这个函数时，把 work 提交给 work queue 就可以了：

```
schedule_work(&aer_recover_work);
```

上述函数会把 work 提供给系统默认的 work queue：system_wq，它是一个队列。

c. 谁来执行 work 中的函数？

不用我们管，schedule_work 函数不仅仅是把 work 放入队列，还会把 kworker 线程唤醒。此线程抢到时间运行时，它就会从队列中取出 work，执行里面的函数。

d. 谁把 work 提交给 work queue？

在中断场景中，可以在中断上半部调用 schedule_work 函数。

总结：

a. 很耗时的中断处理，应该放到线程里去

b. 可以使用 work、work queue

c. 在中断上半部调用 schedule_work 函数，触发 work 的处理

d. 既然是在线程中运行，那对应的函数可以休眠。

18.2.7 新技术：threaded irq

使用线程来处理中断，并不是什么新鲜事。使用 work 就可以实现，但是需要定义 work、调用 schedule_work，好麻烦啊。

太懒了太懒了，就这 2 步你们都不愿意做。

好，内核是为懒人服务的，再杀出一个函数：

```
extern int __must_check
request_threaded_irq(unsigned int irq, irq_handler_t handler,
                     irq_handler_t thread_fn,
                     unsigned long flags, const char *name, void *dev);
```

哪个中断？　　上半部函数，可以为空

在线程里运行的函数

你可以只提供 thread_fn，系统会为这个函数创建一个内核线程。发生中断时，内核线程就会执行这个函数。

说你懒是开玩笑，内核开发者也不会那么在乎懒人。

以前用 work 来线程化地处理中断，一个 worker 线程只能由一个 CPU 执行，多个中断的 work 都由同一个 worker 线程来处理，在单 CPU 系统中也只能忍着了。但是在 SMP 系统中，明明有那么多 CPU 空着，你偏偏让多个中断挤在这个 CPU 上？

新技术 threaded irq，为每一个中断都创建一个内核线程；多个中断的内核线程可以分配到多个 CPU 上执行，这提高了效率。

18.3 重要数据结构

18.4 编程

18.2.1 Linux 的中断不能嵌套

参考文档：

a. 内核 Documentation\devicetree\bindings\Pinctrl\ 目录下：

```
Pinctrl-bindings.txt
fsl,imx-Pinctrl.txt、fsl,imx6ul-Pinctrl.txt
rockchip,Pinctrl.txt
ti,omap-Pinctrl.txt
```

b. 内核 Documentation\gpio 目录下：

```
Pinctrl-bindings.txt
fsl,imx-Pinctrl.txt、fsl,imx6ul-Pinctrl.txt
rockchip,Pinctrl.txt
```

c. 内核 Documentation\devicetree\bindings\gpio 目录下：

```
gpio.txt
```

2.6.30 引入 threaded irq

devm_request_any_context_irq
request_any_context_irq
request_threaded_irq
request_irq

参考资料

中断处理不能嵌套：
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e58aa3d2d0cc

genirq: add threaded interrupt handler support
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3aa551c9b4c40018f0e261a178e3d25478dc04a9

Linux RT(2) – 硬实时 Linux(RT-Preempt Patch)的中断线程化
https://www.veryarm.com/110619.html

Linux 中断管理 (1)Linux 中断管理机制
https://www.cnblogs.com/arnoldlu/p/8659981.html