第 19 章 驱动程序基石

19.1 休眠与唤醒

19.1.1 适用场景

在前面引入中断时，我们曾经举过一个例子：

妈妈怎么知道卧室里小孩醒了？

① 时不时进房间看一下：查询方式

简单，但是累

② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：休眠-唤醒

不累，但是妈妈干不了活了

③ 妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟：poll 方式

要浪费点时间，但是可以继续干活。

妈妈要么是被小孩吵醒，要么是被闹钟吵醒。

④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：异步通知

妈妈、小孩互不耽误

当应用程序必须等待某个事件发生，比如必须等待按键被按下时，**可以使用“休眠-唤醒”机制**：

① APP 调用 read 等函数试图读取数据，比如读取按键；

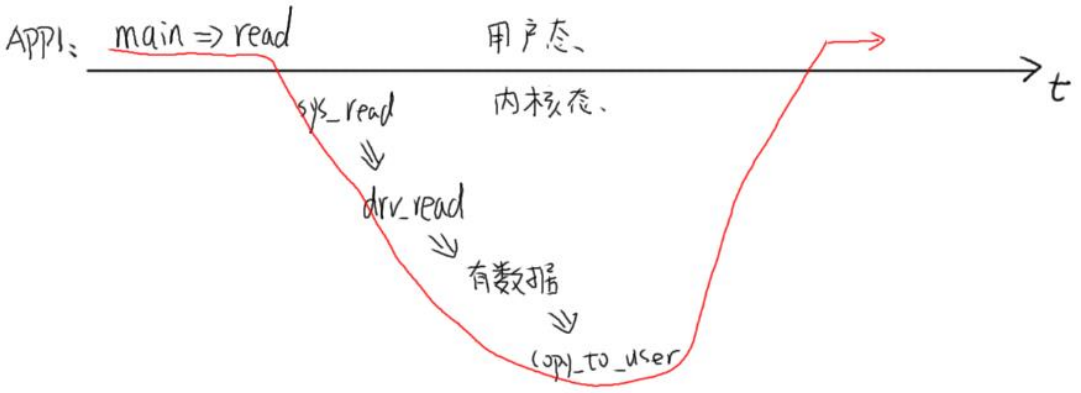
② APP 进入内核态，也就是调用驱动中的对应函数，发现有数据则复制到用户空间并马上返回；

③ 如果 APP 在内核态，也就是在驱动程序中发现没有数据，则 APP 休眠；

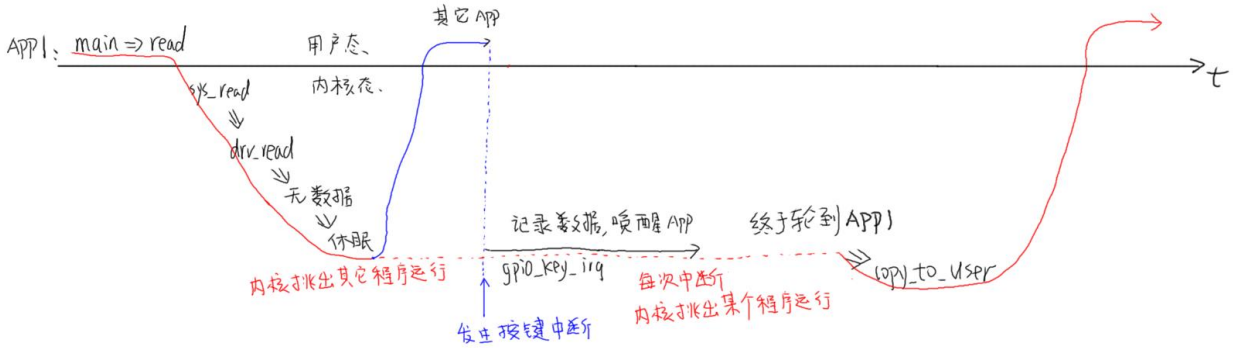
④ 当有数据时，比如当按下按键时，驱动程序的中断服务程序被调用，它会记录数据、唤醒 APP；

⑤ APP 继续运行它的内核态代码，也就是驱动程序中的函数，复制数据到用户空间并马上返回。

驱动中有数据时，下图中红线就是 APP1 的执行过程，涉及用户态、内核态：



驱动中没有数据时，APP1 在内核态执行到 drv_read 时会休眠。所谓休眠就是把自己的状态改为非 RUNNING，这样内核的调度器就不会让它运行。当按下按键，驱动程序中的中断服务程序被调用，它会记录数据，并唤醒 APP1。所以唤醒就是把程序的状态改为 RUNNING，这样内核的调度器有合适的时间就会让它运行。当 APP1 再次运行时，就会继续执行 drv_read 中剩下的代码，把数据复制回用户空间，返回用户空间。APP1 的执行过程如下图的红色实线所示，它被分成了 2 段：



值得注意的是，上面 2 个图中红线部分都属于 APP1 的“上下文”，或者这样说：红线所涉及的代码，都是 APP1 调用的。但是按键的中断服务程序，不属于 APP1 的“上下文”，这是突如其来的，当中断发生时，APP1 正在休眠呢。

在 APP1 的“上下文”，也就是在 APP1 的执行过程中，它是可以休眠的。

在中断的处理过程中，也就是 gpio_key_irq 的执行过程中，它不能休眠：“中断”怎么能休眠？“中断”休眠了，谁来调度其他 APP 啊？

所以，请记住：**在中断处理函数中，不能休眠**，也就不能调用会导致休眠的函数。

19.1.2 内核函数

19.1.2.1 休眠函数

参考内核源码：include\linux\wait.h。

函数	说明
wait_event_interruptible(wq, condition)	休眠，直到 condition 为真； 休眠期间是可被打断的，可以被信号打断
wait_event(wq, condition)	休眠，直到 condition 为真； 退出的唯一条件是 condition 为真，信号也不好使
wait_event_interruptible_timeout(wq, condition, timeout)	休眠，直到 condition 为真或超时； 休眠期间是可被打断的，可以被信号打断
wait_event_timeout(wq, condition, timeout)	休眠，直到 condition 为真； 退出的唯一条件是 condition 为真，信号也不好使

比较重要的参数就是：

① wq：waitqueue，等待队列

休眠时除了把程序状态改为非 RUNNING 之外，还要把进程/进程放入 wq 中，以后中断服务程序要从 wq 中把它取出来唤醒。

没有 wq 的话，茫茫人海中，中断服务程序去哪里找到你？

② condition

这可以是一个变量，也可以是任何表达式。表示“一直等待，直到 condition 为真”。

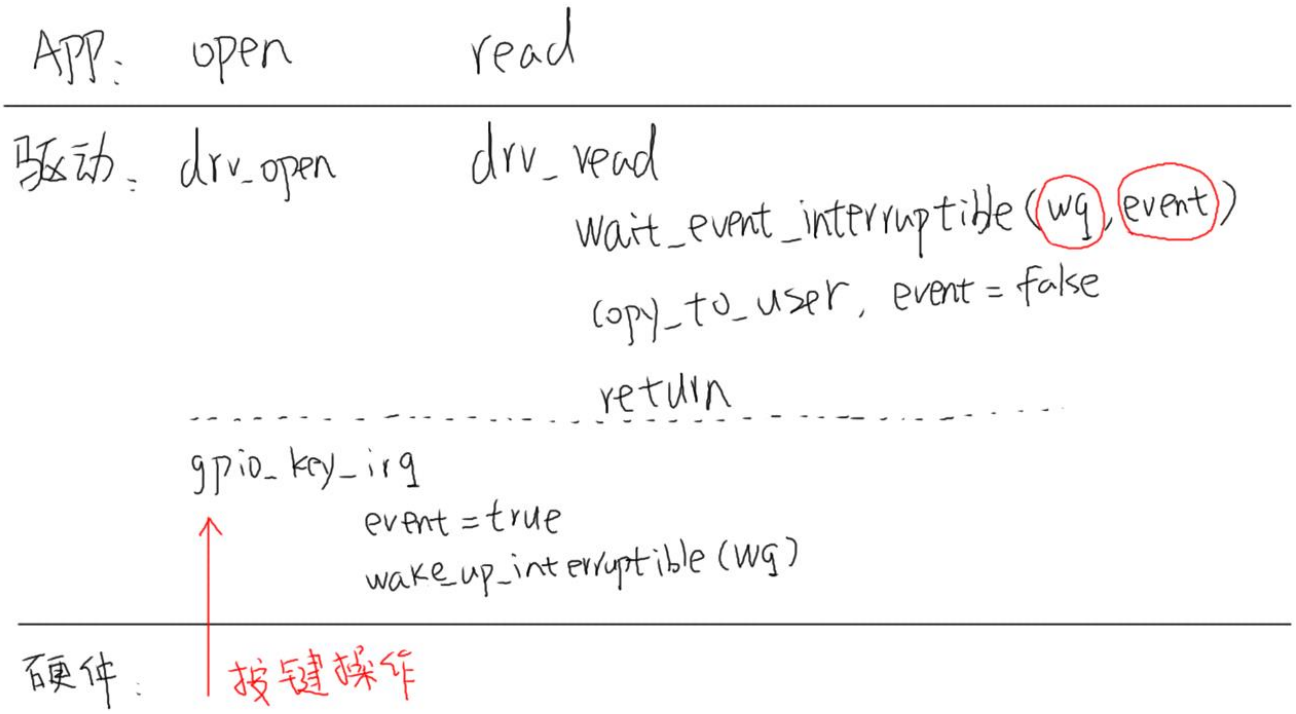
19.1.2.2 唤醒函数

参考内核源码：include\linux\wait.h。

函数	说明
wake_up_interruptible(x)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”的线程，只唤醒其中的一个线程
wake_up_interruptible_nr(x, nr)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”的线程，只唤醒其中的 nr 个线程
wake_up_interruptible_all(x)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”的线程，唤醒其中的所有线程
wake_up(x)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”或“TASK_UNINTERRUPTIBLE”的线程，只唤醒其中的一个线程
wake_up_nr(x, nr)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”或“TASK_UNINTERRUPTIBLE”的线程，只唤醒其中 nr 个线程
wake_up_all(x)	唤醒 x 队列中状态为“TASK_INTERRUPTIBLE”或“TASK_UNINTERRUPTIBLE”的线程，唤醒其中的所有线程

19.1.3 驱动框架

驱动框架如下：



要休眠的线程，放在 wq 队列里，中断处理函数从 wq 欠列里把它取出来唤醒。

所以，我们要做这几件事：

① 初始化 wq 队列

② 在驱动的 read 函数中，调用 wait_event_interruptible：

它本身会判断 event 是否为 FALSE，如果为 FASLE 表示无数据，则休眠。

当从 wait_event_interruptible 返回后，把数据复制回用户空间。

③ 在中断服务程序里：

设置 event 为 TRUE，并调用 wake_up_interruptible 唤醒线程。

19.1.4 编程

使用 GIT 命令载后，源码位于这个目录下：

```
01_all_series_quickstart\
    04_快速入门_正式开始\
        02_嵌入式 Linux 驱动开发基础知识\source\
            06_gpio_irq\
                02_read_key_irq\ 和 03_read_key_irq_circle_buffer
```

03_read_key_irq_circle_buffer 使用了环型缓冲区，可以避免按键丢失。

19.1.4.1 驱动程序关键代码

02_read_key_irq\gpio_key_drv.c 中，要先定义“wait queue”：

```
41 static DECLARE_WAIT_QUEUE_HEAD(gpio_key_wait);
```

在驱动的读函数里调用 wait_event_interruptible：

```
44 static ssize_t gpio_key_drv_read (struct file *file, char __user *buf, size_t size, loff_t
*offset)
45 {
46      //printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
47      int err;
48
49      wait_event_interruptible(gpio_key_wait, g_key);
50      err = copy_to_user(buf, &g_key, 4);
51      g_key = 0;
52
53      return 4;
54 }
```

第 49 行并不一定会进入休眠，它会先判断 g_key 是否为 TRUE。

执行到第 50 行时，表示要么有了数据(g_key 为 TRUE)，要么有信号等待处理(本节课程不涉及信号)。

假设 g_key 等于 0，那么 APP 会执行到上述代码第 49 行时进入休眠状态。它被谁唤醒？被控制的中断服务程序：

```
64 static irqreturn_t gpio_key_isr(int irq, void *dev_id)
65 {
66      struct gpio_key *gpio_key = dev_id;
67      int val;
68      val = gpiod_get_value(gpio_key->gpiod);
69
70
71      printk("key %d %d\n", gpio_key->gpio, val);
72      g_key = (gpio_key->gpio << 8) | val;
73      wake_up_interruptible(&gpio_key_wait);
74
75      return IRQ_HANDLED;
76 }
```

上述代码中，第 72 行确定按键值 g_key，g_key 也就变为 TRUE 了。

然后在第 73 行唤醒 gpio_key_wait 中的第 1 个线程。

注意这 2 个函数，一个没有使用“&”，另一个使用了“&”：

```
wait_event_interruptible(gpio_key_wait, g_key);
wake_up_interruptible(&gpio_key_wait);
```

19.1.4.1 应用程序

应用程序并不复杂，调用 open、read 即可，代码在 button_test.c 中：

```
25      /* 2. 打开文件 */
26      fd = open(argv[1], O_RDWR);
27      if (fd == -1)
28      {
29              printf("can not open file %s\n", argv[1]);
30              return -1;
31      }
32
33      while (1)
34      {
35              /* 3. 读文件 */
36              read(fd, &val, 4);
37              printf("get button : 0x%x\n", val);
38      }
```

在 33 行～38 行的循环中，APP 基本上都是休眠状态。你可以执行 top 命令查看 CPU 占用率。

19.1.5 上机实验

跟上一节视频类似，**需要先修改设备树，请使用上一节视频的设备树文件**。

然后安装驱动程序，运行测试程序。

```
# insmod -f gpio_key_drv.ko
# ls /dev/100ask_gpio_key
/dev/100ask_gpio_key
# ./button_test /dev/100ask_gpio_key  &
# top
```

19.1.6 使用环形缓冲区改进驱动程序

使用 GIT 命令载后，源码位于这个目录下：

```
01_all_series_quickstart\
    04_快速入门_正式开始\
        02_嵌入式 Linux 驱动开发基础知识\source\
            06_gpio_irq\
                03_read_key_irq_circle_buffer
```

使用环形缓冲区，可以在一定程序上避免按键数据丢失，关键代码如下：

```
39: /* 环形缓冲区 */
40: #define BUF_LEN 128
41: static int g_keys[BUF_LEN];
42: static int r, w;                    // r,w是读写位置
43:
44: #define NEXT_POS(x) ((x+1) % BUF_LEN)
45:
46: static int is_key_buf_empty(void)
47: {
48:     return (r == w);   // 一开始r,w都是0, r==w表示空
49: }
50:
51: static int is_key_buf_full(void)
52: {
53:     return (r == NEXT_POS(w));    // 下一个写的位置等于r，表示满
54: }                                  // 容量为128的buffer,
55:                                    // 存有127个数据时我们就认为满了
56: static void put_key(int key)
57: {
58:     if (!is_key_buf_full())
59:     {
60:         g_keys[w] = key;          // 把数据放入w位置
61:         w = NEXT_POS(w);          // 移动w
62:     }
63: }
64:
65: static int get_key(void)
66: {
67:     int key = 0;
68:     if (!is_key_buf_empty())
69:     {
70:         key = g_keys[r];          // 从r位置读数据
71:         r = NEXT_POS(r);          // 移动r
72:     }
73:     return key;
74: }
75:
```

使用环形缓冲区之后，休眠函数可以这样写：

```
86      wait_event_interruptible(gpio_key_wait, !is_key_buf_empty());
87      key = get_key();
88      err = copy_to_user(buf, &key, 4);
```

唤醒函数可以这样写：

```
111     key = (gpio_key->gpio << 8) | val;
112     put_key(key);
113     wake_up_interruptible(&gpio_key_wait);
```

19.2 POLL 机制

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
    04_快速入门_正式开始\
        02_嵌入式 Linux 驱动开发基础知识\source\
            06_gpio_irq\
                04_read_key_irq_poll
```

19.2.1 适用场景

在前面引入中断时，我们曾经举过一个例子：

妈妈怎么知道卧室里小孩醒了？

① 时不时进房间看一下：查询方式

简单，但是累

② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：休眠-唤醒

不累，但是妈妈干不了活了

③ 妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟：poll 方式

要浪费点时间，但是可以继续干活。

妈妈要么是被小孩吵醒，要么是被闹钟吵醒。

④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：异步通知

妈妈、小孩互不耽误

使用休眠-唤醒的方式等待某个事件发生时，有一个缺点：等待的时间可能很久。我们可以加上一个超时时间，这时就可以使用 poll 机制。

① APP 不知道驱动程序中是否有数据，可以先调用 poll 函数查询一下，poll 函数可以传入超时时间；

② APP 进入内核态，调用到驱动程序的 poll 函数，如果有数据的话立刻返回；

③ 如果发现没有数据时就休眠一段时间；

④ 当有数据时，比如当按下按键时，驱动程序的中断服务程序被调用，它会记录数据、唤醒 APP；

⑤ 当超时时间到了之后，内核也会唤醒 APP；

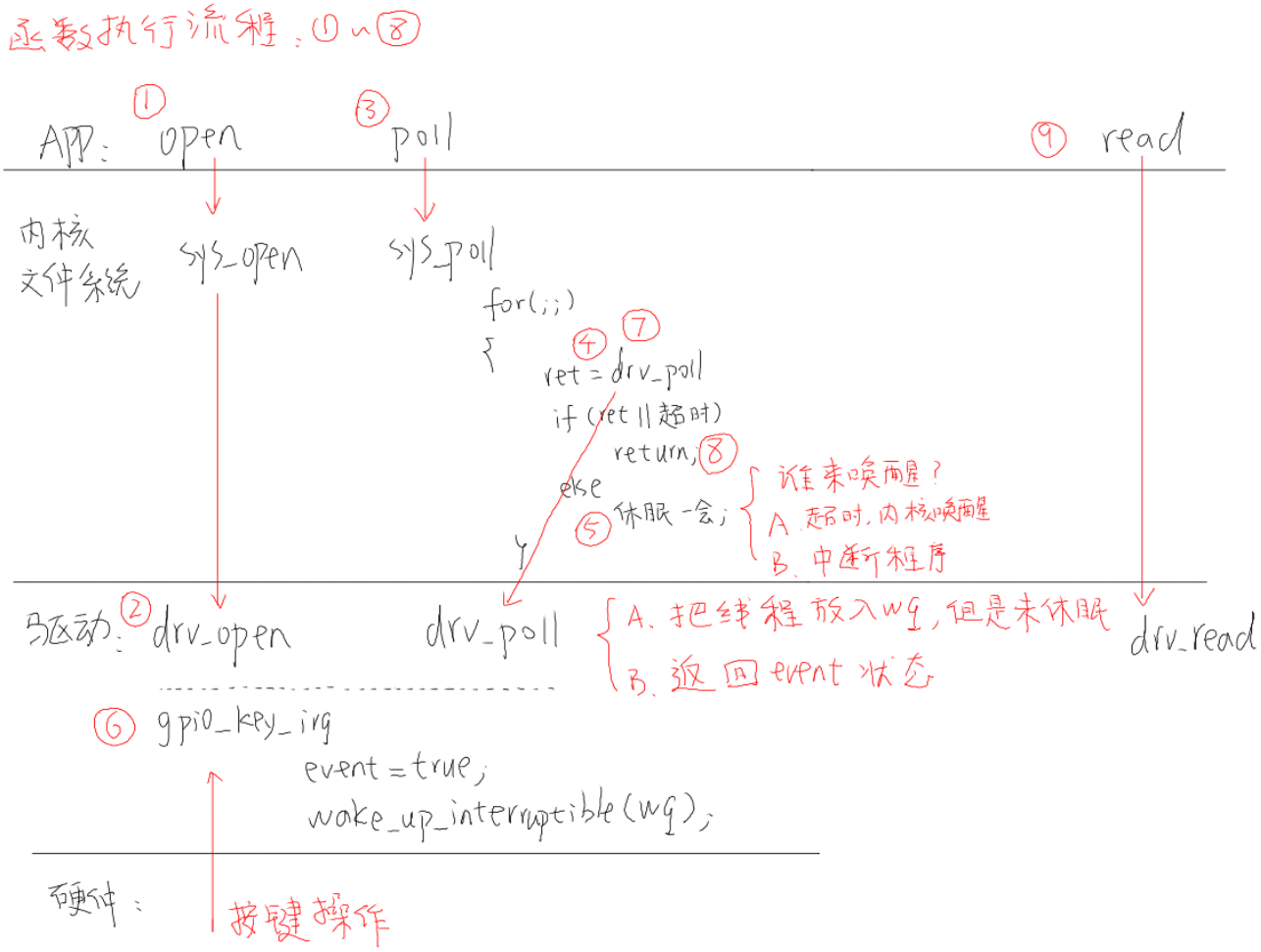
⑥ APP 根据 poll 函数的返回值就可以知道是否有数据，如果有数据就调用 read 得到数据

19.2.2 使用流程

妈妈进入房间时，会先看小孩醒没醒，闹钟响之后走出房间之前又会再看小孩醒没醒。

注意：看了 2 次小孩！

POLL 机制也是类似的，流程如下：



函数执行流程如上图①~⑧所示，重点从③开始看。假设一开始无按键数据：

③ APP 调用 poll 之后，进入内核态；

④ 导致驱动程序的 drv_poll 被调用：

注意，drv_poll 要把自己这个线程挂入等待队列 wq 中；假设不放入队列里，那以后发生中断时，中断服务程序去哪里找到你嘛？

drv_poll 还会判断一下：有没有数据啊？返回这个状态。

⑤ 假设当前没有数据，则休眠一会；

⑥ 在休眠过程中，按下了按键，发生了中断：

在中断服务程序里记录了按键值，并且从 wq 中把线程唤醒了。

⑦ 线程从休眠中被唤醒，继续执行 for 循环，再次调用 drv_poll：

drv_poll 返回数据状态

⑧ 哦，你有数据，那从内核态返回到应用态吧

⑨ APP 调用 read 函数读数据

如果一直没有数据，调用流程也是类似的，重点从③开始看，如下：

③ APP 调用 poll 之后，进入内核态；

④ 导致驱动程序的 drv_poll 被调用：

注意，drv_poll 要把自己这个线程挂入等待队列 wq 中；假设不放入队列里，那以后发生中断时，中断服务程序去哪里找到你嘛？

drv_poll 还会判断一下：有没有数据啊？返回这个状态。

⑤ 假设当前没有数据，则休眠一会；

⑥ 在休眠过程中，一直没有按下了按键，超时时间到：内核把这个线程唤醒；

⑦ 线程从休眠中被唤醒，继续执行 for 循环，再次调用 drv_poll：

drv_poll 返回数据状态

⑧ 哦，你还是没有数据，但是超时时间到了，那从内核态返回到应用态吧

⑨ APP 不能调用 read 函数读数据

注意几点：

① drv_poll 要把线程挂入队列 wq，但是并不是在 drv_poll 中进入休眠，而是在调用 drv_poll 之后休眠

② drv_poll 要返回数据状态

③ APP 调用一次 poll，有可能会导致 drv_poll 被调用 2 次

④ 线程被唤醒的原因有 2：中断发生了去队列 wq 中把它唤醒，超时时间到了内核把它唤醒

⑤ APP 要判断 poll 返回的原因：有数据，还是超时。有数据时再去调用 read 函数。

19.2.3 驱动编程

使用 poll 机制时，驱动程序的核心就是提供对应的 drv_poll 函数。

在 drv_poll 函数中要做 2 件事：

① 把当前线程挂入队列 wq：poll_wait

APP 调用一次 poll，可能导致 drv_poll 被调用 2 次，但是我们并不需要把当前线程挂入队列 2 次。

可以使用内核的函数 poll_wait 把线程挂入队列，如果线程已经在队列里了，它就不会再次挂入。

② 返回设备状态：

APP 调用 poll 函数时，有可能是查询“有没有数据可以读”：POLLIN，也有可能是查询“你有没有空间给我写数据”：POLLOUT。

所以 drv_poll 要返回自己的当前状态：(POLLIN | POLLRDNORM) 或 (POLLOUT | POLLWRNORM)。

POLLRDNORM 等同于 POLLIN，为了兼容某些 APP 把它们一起返回。

POLLWRNORM 等同于 POLLOUT ，为了兼容某些 APP 把它们一起返回。

APP 调用 poll 后，很有可能会休眠。对应的，在按键驱动的中断服务程序中，也要有唤醒操作。

驱动程序中 poll 的代码如下：

```
static unsigned int gpio_key_drv_poll(struct file *fp, poll_table * wait)
{
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    poll_wait(fp, &gpio_key_wait, wait);
    return is_key_buf_empty() ? 0 : POLLIN | POLLRDNORM;
}
```

19.2.4 应用编程

注意：APP 可以调用 poll 或 select 函数，这 2 个函数的作用是一样的。

poll/select 函数可以监测多个文件，可以监测多种事件：

事件类型	说明
POLLIN	有数据可读
POLLRDNORM	等同于 POLLIN
POLLRDBAND	Priority band data can be read，有优先级较较高的“band data”可读 Linux 系统中很少使用这个事件
POLLPRI	高优先级数据可读
POLLOUT	可以写数据
POLLWRNORM	等同于 POLLOUT
POLLWRBAND	Priority data may be written
POLLERR	发生了错误
POLLHUP	挂起
POLLNVAL	无效的请求，一般是 fd 未 open

在调用 poll 函数时，要指明：

① 你要监测哪一个文件：哪一个 fd

② 你想监测这个文件的哪种事件：是 POLLIN、还是 POLLOUT

最后，在 poll 函数返回时，要判断状态。

应用程序代码如下：

```
struct pollfd fds[1];
int timeout_ms = 5000;
int ret;

fds[0].fd = fd;
fds[0].events = POLLIN;

ret = poll(fds, 1, timeout_ms);
if ((ret == 1) && (fds[0].revents & POLLIN))
{
    read(fd, &val, 4);
    printf("get button : 0x%x\n", val);
}
```

19.2.5 现场编程

19.2.6 上机实验

19.2.7 POLL 机制的内核代码详解

Linux APP 系统调用，基本都可以在它的名字前加上“sys_”前缀，这就是它在内核中对应的函数。比如系统调用 open、read、write、poll，与之对应的内核函数为：sys_open、sys_read、sys_write、sys_poll。

对于系统调用 poll 或 select，它们对应的内核函数都是 sys_poll。分析 sys_poll，即可理解 poll 机制。

19.2.7.1 sys_poll 函数

sys_poll位于fs/select.c文件中，代码如下：

```
SYSCALL_DEFINE3(poll, struct pollfd __user *, ufds, unsigned int, nfds,
                int, timeout_msecs)
{
        struct timespec64 end_time, *to = NULL;
        int ret;

        if (timeout_msecs >= 0) {
                to = &end_time;
                poll_select_set_timeout(to, timeout_msecs / MSEC_PER_SEC,
                        NSEC_PER_MSEC * (timeout_msecs % MSEC_PER_SEC));
        }

        ret = do_sys_poll(ufds, nfds, to);
……
```

SYSCALL_DEFINE3 是一个宏，它定义于 include/linux/syscalls.h，展开后就有 sys_poll 函数。

sys_poll 对超时参数稍作处理后，直接调用 do_sys_poll。

19.2.7.2 do_sys_poll 函数

do_sys_poll 位于 fs/select.c 文件中，我们忽略其他代码，只看关键部分：

```
int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds,
                struct timespec64 *end_time)
{
……
        poll_initwait(&table);
        fdcount = do_poll(head, &table, end_time);
        poll_freewait(&table);
……
}
```

poll_initwait 函数非常简单，它初始化一个 poll_wqueues 变量 table：

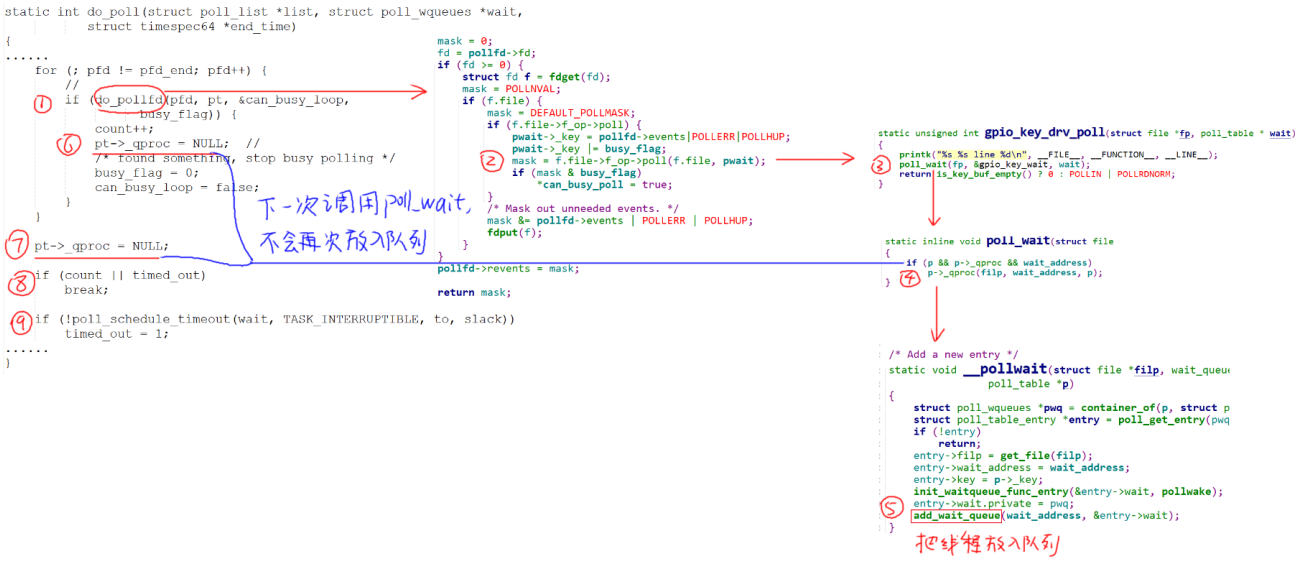
```
poll_initwait
    init_poll_funcptr(&pwq->pt, __pollwait);
        pt->qproc = qproc;
```

即 table->pt->qproc = __pollwait，__pollwait 将在驱动的 poll 函数里用到。

do_poll 函数才是核心，继续看代码。

19.2.7.3 do_poll 函数

do_poll 函数位于 fs/select.c 文件中，这是 POLL 机制中最核心的代码，贴图如下：



① 从这里开始，将会导致驱动程序的 poll 函数被第一次调用。

沿着②③④⑤，你可以看到：驱动程序里的 poll_wait 会调用__pollwait 函数把线程放入某个队列。

当执行完①之后，在⑥或⑦处，pt->_qproc 被设置为 NULL，所以第二次调用驱动程序的 poll 时，不会再次把线程放入某个队列里。

⑧ 如果驱动程序的 poll 返回有效值，则 count 非 0，跳出循环；

⑨ 否则休眠一段时间；当休眠时间到，或是被中断唤醒时，会再次循环、再次调用驱动程序的 poll。

回顾 APP 的代码，APP 可以指定“想等待某些事件”，poll 函数返回后，可以知道“发生了哪些事件”：

```
fds[0].fd = fd;
fds[0].events = POLLIN;

while (1)
{
    /* 3. 读文件 */
    ret = poll(fds, 1, timeout_ms);
    if ((ret == 1) && (fds[0].revents & POLLIN))
    {
        read(fd, &val, 4);
        printf("get button : 0x%x\n", val);
    }
```

想等待什么事件
得到了什么事件

驱动程序里怎么体现呢？在上上一个图中，看②位置处，细说如下：

```
    if (f.file) {
        mask = DEFAULT_POLLMASK;
        if (f.file->f_op->poll) {
            pwait->_key = pollfd->events|POLLERR|POLLHUP;
            pwait->_key |= busy_flag;
            mask = f.file->f_op->poll(f.file, pwait);
            if (mask & busy_flag)
                *can_busy_poll = true;
        }
        /* Mask out unneeded events. */
        mask &= pollfd->events | POLLERR | POLLHUP;
        fdput(f);
    }
}
pollfd->revents = mask;
```

1. 驱动程序返回的状态
2. 是否是APP期待的？
3. 写入revents

19.3 异步通知

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
    04_快速入门_正式开始\
        02_嵌入式 Linux 驱动开发基础知识\source\
            06_gpio_irq\
                05_read_key_irq_poll_fasync
```

19.3.1 适用场景

在前面引入中断时，我们曾经举过一个例子：

妈妈怎么知道卧室里小孩醒了？

① 时不时进房间看一下：查询方式

简单，但是累

② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：休眠-唤醒

不累，但是妈妈干不了活了

③ 妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟：poll 方式

要浪费点时间，但是可以继续干活。

妈妈要么是被小孩吵醒，要么是被闹钟吵醒。

④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：异步通知

妈妈、小孩互不耽误

使用休眠-唤醒、POLL 机制时，都需要休眠等待某个事件发生时，它们的差别在于后者可以指定休眠的时长。

在现实生活中：妈妈可以不陪小孩睡觉，小孩醒了之后可以主动通知妈妈。

如果 APP 不想休眠怎么办？也有类似的方法：驱动程序有数据时主动通知 APP，APP 收到信号后执行信息处理函数。

什么叫“异步通知”？

你去买奶茶：

你在旁边等着，眼睛盯着店员，生怕别人插队，他一做好你就知道：你是主动等待他做好，这叫“同步”。

你付钱后就去玩手机了，店员做好后他会打电话告诉你：你是被动获得结果，这叫“异步”。

19.3.2 使用流程

驱动程序怎么通知 APP：发信号，这只有 3 个字，却可以引发很多问题：

① 谁发：驱动程序发

② 发什么：信号

③ 发什么信号：SIGIO

④ 怎么发：内核里提供有函数

⑤ 发给谁：APP，APP 要把自己告诉驱动
⑥ APP 收到后做什么：执行信号处理函数
⑦ 信号处理函数和信号，之间怎么挂钩：APP 注册信号处理函数

小孩通知妈妈的事情有很多：饿了、渴了、想找人玩。

Linux 系统中也有很多信号，在 Linux 内核源文件 include\uapi\asm-generic\signal.h 中，有很多信号的宏定义：

```
#define SIGHUP       1
#define SIGINT       2
#define SIGQUIT      3
#define SIGILL       4
#define SIGTRAP      5
#define SIGABRT      6
#define SIGIOT       6
#define SIGBUS       7
#define SIGFPE       8
#define SIGKILL      9
#define SIGUSR1     10
#define SIGSEGV     11
#define SIGUSR2     12
#define SIGPIPE     13
#define SIGALRM     14
#define SIGTERM     15
#define SIGSTKFLT   16
#define SIGCHLD     17
#define SIGCONT     18
#define SIGSTOP     19
#define SIGTSTP     20
#define SIGTTIN     21
#define SIGTTOU     22
#define SIGURG      23
#define SIGXCPU     24
#define SIGXFSZ     25
#define SIGVTALRM   26
#define SIGPROF     27
#define SIGWINCH    28
#define SIGIO       29   <-- 驱动常用信号，表示有IO事件
#define SIGPOLL     SIGIO
```

就 APP 而言，你想处理 SIGIO 信息，那么需要提供信号处理函数，并且要跟 SIGIO 挂钩。这可以通过一个 signal 函数来“给某个信号注册处理函数”，用法如下：

```
#include <signal.h>

typedef void (*sighandler_t)(int); // 1. 先编写函数

sighandler_t signal(int signum, sighandler_t handler); // 2. 注册
                    哪个信号？      信号处理函数
```

APP 还要做什么事？想想这几个问题：

① 内核里有那么多驱动，你想让哪一个驱动给你发 SIGIO 信号？
APP 要打开驱动程序的设备节点。

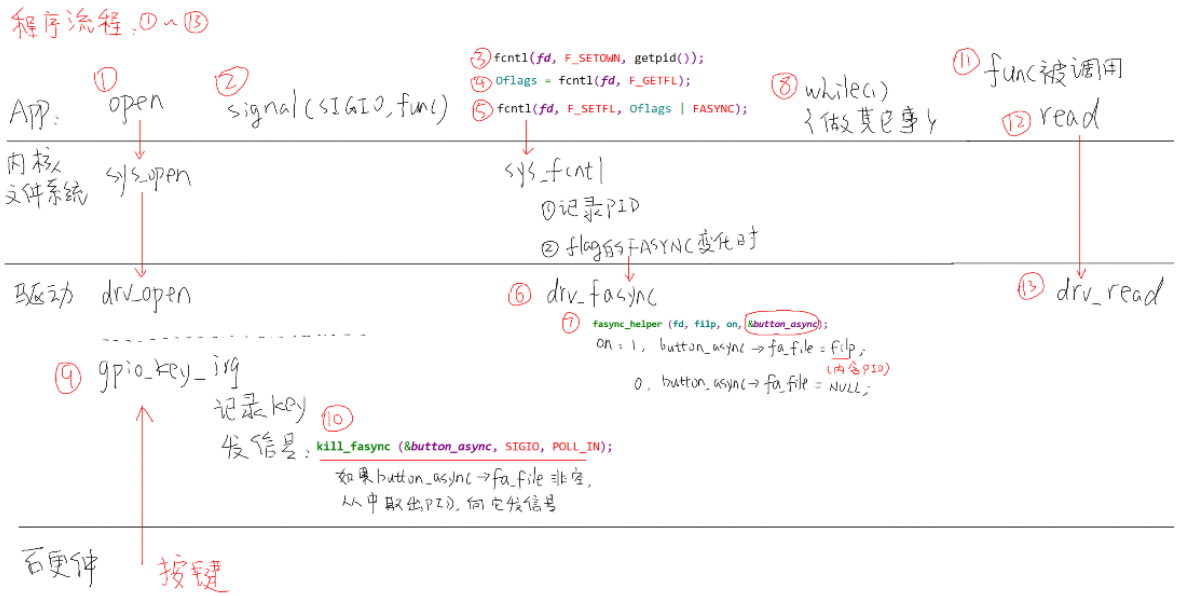
② 驱动程序怎么知道要发信号给你而不是别人？
APP 要把自己的进程 ID 告诉驱动程序。

③ APP 有时候想收到信号，有时候又不想收到信号：
应该可以把 APP 的意愿告诉驱动。
驱动程序要做什么？发信号。

① APP 设置进程 ID 时，驱动程序要记录下进程 ID；
② APP 还要使能驱动程序的异步通知功能，驱动中有对应的函数：
APP 打开驱动程序时，内核会创建对应的 file 结构体，file 中有 f_flags；
f_flags 中有一个 FASYNC 位，它被设置为 1 时表示使能异步通知功能。
当 f_flags 中的 FASYNC 位发生变化时，驱动程序的 fasync 函数被调用。
③ 发生中断时，有数据时，驱动程序调用内核辅助函数发信号。
这个辅助函数名为 kill_fasync。

完美！
APP 收到信号后，是怎么执行信号处理函数的？
这个，很难，有兴趣的话就看本节最后的文档。初学者没必要看。

综上所述，使用异步通知，也就是使用信号的流程如下图所示：



重点从②开始：
② APP 给 SIGIO 这个信号注册信号处理函数 func，以后 APP 收到 SIGIO 信号时，这个函数会被自动调用；
③ 把 APP 的 PID(进程 ID)告诉驱动程序，这个调用不涉及驱动程序，在内核的文件系统层次记录 PID；
④ 读取驱动程序文件 Flag；
⑤ 设置 Flag 里面的 FASYNC 位为 1：当 FASYNC 位发生变化时，会导致驱动程序的 fasync 被调用；
⑥⑦ 调用 faync_helper，它会根据 FAYSNC 的值决定是否设置 button_async->fa_file=驱动文件 filp：
驱动文件 filp 结构体里面含有之前设置的 PID。
⑧ APP 可以做其他事；
⑨⑩ 按下按键，发生中断，驱动程序的中断服务程序被调用，里面调用 kill_fasync 发信号；
⑪⑫⑬ APP 收到信号后，它的信号处理函数被自动调用，可以在里面调用 read 函数读取按键。

19.3.3 驱动编程

使用异步通知时，驱动程序的核心有 2：
① 提供对应的 drv_fasync 函数；
② 并在合适的时机发信号。

drv_fasync 函数很简单，调用 fasync_helper 函数就可以，如下：

```
static struct fasync_struct *button_async;
static int drv_fasync (int fd, struct file *filp, int on)
{
    return fasync_helper (fd, filp, on, &button_async);
}
```

fasync_helper 函数会分配、构造一个 fasync_struct 结构体 button_async：

① 驱动文件的 flag 被设置为 FAYNC 时：

```
button_async->fa_file = filp;  // filp 表示驱动程序文件，里面含有之前设置的 PID
```

② 驱动文件被设置为非 FASYNC 时：

```
button_async->fa_file = NULL;
```

以后想发送信号时，使用 button_async 作为参数就可以，它里面“可能”含有 PID。

什么时候发信号呢？在本例中，在 GPIO 中断服务程序中发信号。

怎么发信号呢？代码如下：

```
kill_fasync (&button_async, SIGIO, POLL_IN);
```

第 1 个参数：button_async->fa_file 非空时，可以从中得到 PID，表示发给哪一个 APP；

第 2 个参数表示发什么信号：SIGIO；

第 3 个参数表示为什么发信号：POLL_IN，有数据可以读了。(APP 用不到这个参数)

19.3.4 应用编程

应用程序要做的事情有这几件：

① 编写信号处理函数：

```
static void sig_func(int sig)
{
    int val;
    read(fd, &val, 4);
    printf("get button : 0x%x\n", val);
}
```

② 注册信号处理函数：

```
signal(SIGIO, sig_func);
```

③ 打开驱动：

```
fd = open(argv[1], O_RDWR);
```

④ 把进程 ID 告诉驱动：

```
fcntl(fd, F_SETOWN, getpid());
```

⑤ 使能驱动的 FASYNC 功能：

```
flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | FASYNC);
```

19.3.5 现场编程

19.3.6 上机编程

19.3.7 异步通知机制内核代码详解

还没写

19.4 阻塞与非阻塞

所谓阻塞，就是等待某件事情发生。比如调用 read 读取按键时，如果没有按键数据则 read 函数不会返回，它会让线程休眠等待。

使用 poll 时，如果传入的超时时间不为 0，这种访问方法也是阻塞的。

使用 poll 时，可以设置超时时间为 0，这样即使没有数据它也会立刻返回，这就是非阻塞方式。能不能让 read 函数既能工作于阻塞方式，也可以工作于非阻塞方式？**可以**！

APP 调用 open 函数时，传入 O_NONBLOCK，就表示要使用非阻塞方式；默认是阻塞方式。

注意：对于普通文件、块设备文件，O_NONBLOCK 不起作用。

注意：对于字符设备文件，O_NONBLOCK 起作用的前提是驱动程序针对 O_NONBLOCK 做了处理。

只能在 open 时表明 O_NONBLOCK 吗？在 open 之后，也可以通过 fcntl 修改为阻塞或非阻塞。

使用 GIT 命令载后，本节源码位于这个目录下：

```
01_all_series_quickstart\
    04_快速入门_正式开始\
        02_嵌入式 Linux 驱动开发基础知识\source\
            06_gpio_irq\
                06_read_key_irq_poll_fasync_block
```

19.4.1 应用编程

open 时设置：

```
int  fd = open(“/dev/xxx”, O_RDWR | O_NONBLOCK);  /* 非阻塞方式 */
int  fd = open(“/dev/xxx”, O_RDWR );  /* 阻塞方式 */
```

open 之后设置：

```
int flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);  /* 非阻塞方式 */
fcntl(fd, F_SETFL, flags & ~O_NONBLOCK);  /* 阻塞方式 */
```

19.4.2 驱动编程

以 drv_read 为例：

```
static ssize_t drv_read(struct file *fp, char __user *buf, size_t count, loff_t *ppos)
 {
    if (queue_empty(&as->queue) && fp->f_flags & O_NONBLOCK)
        return -EAGAIN;

    wait_event_interruptible(apm_waitqueue, !queue_empty(&as->queue));
    ……
}
```

从驱动代码也可以看出来，当 APP 打开某个驱动时，在内核中会有一个 struct file 结构体对应这个驱动，这个结构体中有 f_flags，就是打开文件时的标记位；可以设置 f_flasgs 的 O_NONBLOCK 位，表示非阻塞；也可以清除这个位表示阻塞。

驱动程序要根据这个标记位决定事件未就绪时是休眠和还是立刻返回。

19.4.3 驱动开发原则

驱动程序程序“只提供功能，不提供策略”。就是说驱动程序可以提供休眠唤醒、查询等等各种方式，，驱动程序只提供这些能力，怎么用由 APP 决定。

19.5 定时器

19.5.1 内核函数

所谓定时器，就是闹钟，时间到后你就要做某些事。有 2 个要素：时间、做事，换成程序员的话就是：超时时间、函数。

在内核中使用定时器很简单，涉及这些函数(参考内核源码 include\linux\timer.h)：

① setup_timer(timer, fn, data)：

设置定时器，主要是初始化 timer_list 结构体，设置其中的函数、参数。

② void add_timer(struct timer_list *timer)：

向内核添加定时器。timer->expires 表示超时时间。

当超时时间到达，内核就会调用这个函数：timer->function(timer->data)。

③ int mod_timer(struct timer_list *timer, unsigned long expires):

修改定时器的超时时间，

它等同于：del_timer(timer); timer->expires = expires; add_timer(timer);

但是更加高效。

④ int del_timer(struct timer_list *timer)：

删除定时器。

19.5.2 定时器时间单位

编译内核时，可以在内核源码根目录下用“ls -a”看到一个隐藏文件，它就是内核配置文件。打开后可以看到如下这项：

```
CONFIG_HZ=100
```

这表示内核每秒中会发生 100 次系统滴答中断(tick)，这就像人类的心跳一样，这是 Linux 系统的心跳。每发生一次 tick 中断，全局变量 jiffies 就会累加 1。

CONFIG_HZ=100 表示每个滴答是 10ms。

定时器的时间就是基于 jiffies 的，我们修改超时时间时，一般使用这 2 种方法：

① 在 add_timer 之前，直接修改：

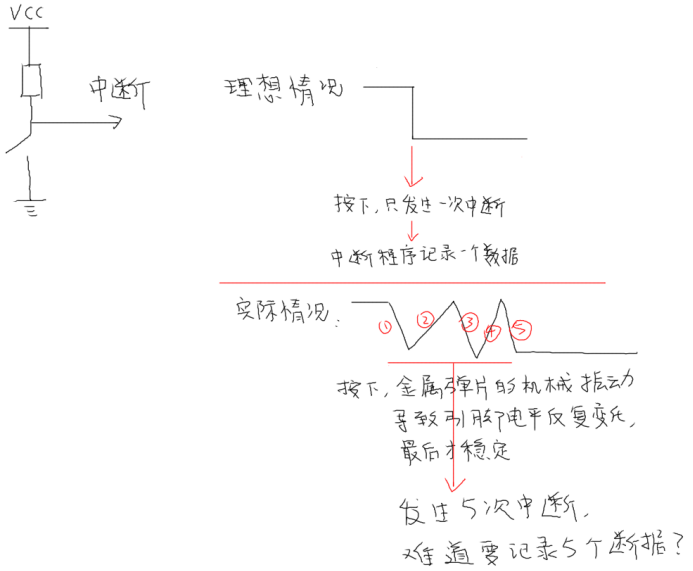
```
timer.expires = jiffies + xxx;   // xxx 表示多少个滴答后超时，也就是 xxx*10ms
timer.expires = jiffies + 2*HZ;  // HZ 等于 CONFIG_HZ，2*HZ 就相当于 2 秒
```

② 在 add_timer 之后，使用 mod_timer 修改：

```
mod_timer(&timer, jiffies + xxx);   // xxx 表示多少个滴答后超时，也就是 xxx*10ms
mod_timer(&timer, jiffies + 2*HZ);  // HZ 等于 CONFIG_HZ，2*HZ 就相当于 2 秒
```

19.5.3 使用定时器处理按键抖动

在实际的按键操作中，可能会有机械抖动：



按下或松开一个按键，它的 GPIO 电平会反复变化，最后才稳定。一般是几十毫秒才会稳定。

如果不处理抖动的话，用户只操作一次按键，中断程序可能会上报多个数据。

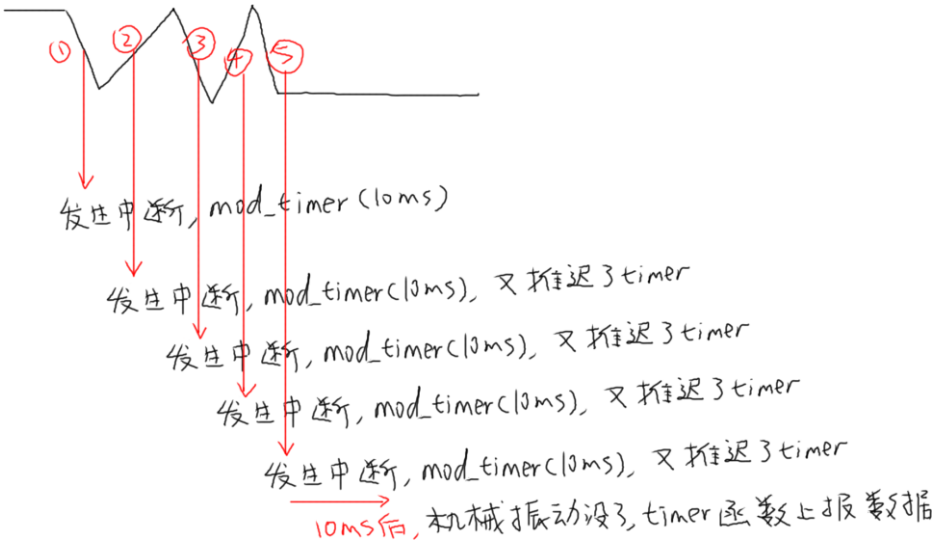
怎么处理？

① 在按键中断程序中，可以循环判断几十毫秒，发现电平稳定之后再上报

② 使用定时器

显然第 1 种方法太耗时，违背“中断要尽快处理”的原则，你的系统会很卡。

怎么使用定时器？看下图：



核心在于：在 GPIO 中断中并不立刻记录按键值，而是修改定时器超时时间，10ms 后再处理。

如果 10ms 内又发生了 GPIO 中断，那就认为是抖动，这时再次修改超时时间为 10ms。

只有 10ms 之内再无 GPIO 中断发生，那么定时器的函数才会被调用。

在定时器函数中记录按键值。

19.5.4 现场编程、上机

19.5.5 深入研究：定时器的内部机制

初学者会用定时器就行，本节不用看。

怎么实现定时器，逻辑上很简单：每发生一次硬件中断时，硬件中断处理完后就会看看有没有软件中断要处理。

定时器就是通过软件中断来实现的，它属于 TIMER_SOFTIRQ 软中断。

对于 TIMER_SOFTIRQ 软中断，初始化代码如下：

```
void __init init_timers(void)
{
    init_timer_cpus();
    init_timer_stats();
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
}
```

当发生硬件中断时，硬件中断处理完后，内核会调用软件中断的处理函数。对于 TIMER_SOFTIRQ，会调用 run_timer_softirq，它的函数如下：

```
run_timer_softirq
    __run_timers(base);
        while (time_after_eq(jiffies, base->clk)) {
            ......
            expire_timers(base, heads + levels);
                fn = timer->function;
                data = timer->data;
                call_timer_fn(timer, fn, data);
                    fn(data);
```

简单地说，add_timer 函数会把 timer 放入内核里某个链表；

在 TIMER_SOFTIRQ 的处理函数中，会从链表中把这些超时的 timer 取出来，执行其中的函数。

怎么判断是否超时？jiffies 大于或等于 timer->expires 时，timer 就超时。

内核中有很多 timer，如果高效地找到超时的 timer？这是比较复杂的，可以看看这文章：

https://blog.csdn.net/tianmohust/article/details/8707162

我们以后如果要深入讲解 timer 的话，会用视频来讲解。

19.5.6 深入研究：找到系统滴答

这只是一些笔记，初学者不用看。

在开发板执行以下命令，可以看到 CPU0 下有一个数值变化特别快，它就是滴答中断：

```
# cat /proc/interrupts
           CPU0
 16:       2532       GPC  55 Level     i.MX Timer Tick
 19:         22       GPC  33 Level     2010000.ecspi
 20:        384       GPC  26 Level     2020000.serial
 21:          0       GPC  98 Level     sai
```

以 100ASK_IMX6ULL 为做，滴答中断名字就是“i.MX Timer Tick”。

在 Linux 内核源码目录下执行以下命令：

```
$ grep "i.MX Timer Tick" * -nr
drivers/clocksource/timer-imx-gpt.c:319:        act->name = "i.MX Timer Tick";
```

打开 timer-imx-gpt.c 319 行左右，可得如下源码：

```
    act->name = "i.MX Timer Tick";
    act->flags = IRQF_TIMER | IRQF_IRQPOLL;
    act->handler = mxc_timer_interrupt;
    act->dev_id = ced;

    return setup_irq(imxtm->irq, act);
```

mxc_timer_interrupt 应该就是滴答中断的处理函数，代码如下：

```
static irqreturn_t mxc_timer_interrupt(int irq, void *dev_id)
{
    struct clock_event_device *ced = dev_id;
    struct imx_timer *imxtm = to_imx_timer(ced);
    uint32_t tstat;

    tstat = readl_relaxed(imxtm->base + imxtm->gpt->reg_tstat);

    imxtm->gpt->gpt_irq_acknowledge(imxtm);

    ced->event_handler(ced);

    return IRQ_HANDLED;
}
```

在上述代码中没看到对 jiffies 的累加操作啊，应该是在 ced->event_handler(ced)中进行。

ced->event_handler(ced)是哪一个函数？不太好找，我使用 QEMU 来调试内核，在 mxc_timer_interrupt 中打断点跟踪代码(以后的课程会讲怎么用 QEMU 调试内核)，发现它对应 tick_handle_periodic。

tick_handle_periodic 位于 kernel\time\tick-common.c 中，它里面的调用关系如下：

```
tick_handle_periodic
    tick_periodic(cpu);
        do_timer(1);
```

```
            jiffies_64 += ticks;  // jiffies 就是 jiffies_64
```

你为何说 jiffies 就是 jiffies_64？在 arch\arm\kernel\vmlinux.lds.S 有如下代码：

```
#ifndef __ARMEB__
jiffies = jiffies_64;
#else
jiffies = jiffies_64 + 4;
#endif
```

上述代码说明了，对于大字节序的 CPU，jiffies 指向 jiffies_64 的高 4 字节；对于小字节序的 CPU，jiffies 指向 jiffies_64 的低 4 字节。

对 jiffies_64 的累加操作，就是对 jiffies 的累加操作。

19.6 中断下半部 tasklet

在前面我们介绍过中断上半部、下半部。中断的处理有几个原则：

① 不能嵌套；

② 越快越好。

在处理当前中断时，即使发生了其他中断，其他中断也不会得到处理，所以中断的处理要越快越好。但是某些中断要做的事情稍微耗时，这时可以把中断拆分为上半部、下半部。

在上半部处理紧急的事情，在上半部的处理过程中，中断是被禁止的；

在下半部处理耗时的事情，在下半部的处理过程中，中断是使能的。

中断上半部、下半部的关系机制，请回顾第 18.2.5 节。

19.6.1 内核函数

1. 定义 tasklet

中断下半部使用结构体 tasklet_struct 来表示，它在内核源码 include\linux\interrupt.h 中定义：

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

其中的 state 有 2 位：

① bit0 表示 TASKLET_STATE_SCHED

等于 1 时表示已经执行了 tasklet_schedule 把该 tasklet 放入队列了；tasklet_schedule 会判断该位，如果已经等于 1 那么它就不会再次把 tasklet 放入队列。

② bit1 表示 TASKLET_STATE_RUN

等于 1 时，表示正在运行 tasklet 中的 func 函数；函数执行完后内核会把该位清 0。

其中的 count 表示该 tasklet 是否使能：等于 0 表示使能了，非 0 表示被禁止了。对于 count 非 0 的 tasklet，里面的 func 函数不会被执行。

使用中断下半部之前，要先实现一个 tasklet_struct 结构体，这可以用这 2 个宏来定义结构体：

```
#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 }
```

使用 DECLARE_TASKLET 定义的 tasklet 结构体，它是使能的；

使用 DECLARE_TASKLET_DISABLED 定义的 tasklet 结构体，它是禁止的；使用之前要先调用 tasklet_enable 使能它。

也可以使用函数来初始化 tasklet 结构体：

```
extern void tasklet_init(struct tasklet_struct *t,
             void (*func)(unsigned long), unsigned long data);
```

2. 使能/禁止 tasklet

```
static inline void tasklet_enable(struct tasklet_struct *t);
static inline void tasklet_disable(struct tasklet_struct *t);
```

tasklet_enable 把 count 增加 1；tasklet_disable 把 count 减 1。

3. 调度 tasklet

```
static inline void tasklet_schedule(struct tasklet_struct *t);
```

把 tasklet 放入链表，并且设置它的 TASKLET_STATE_SCHED 状态为 1。

4. kill tasklet

```
extern void tasklet_kill(struct tasklet_struct *t);
```

如果一个 tasklet 未被调度，tasklet_kill 会把它的 TASKLET_STATE_SCHED 状态清 0；

如果一个 tasklet 已被调度，tasklet_kill 会等待它执行完华，再把它的 TASKLET_STATE_SCHED 状态清 0。

通常在卸载驱动程序时调用 tasklet_kill。

19.6.2 tasklet 使用方法

先定义 tasklet，需要使用时调用 tasklet_schedule，驱动卸载前调用 tasklet_kill。

tasklet_schedule 只是把 tasklet 放入内核队列，它的 func 函数会在软件中断的执行过程中被调用。

19.6.3 tasklet 内部机制

作为初学者，可以不看本节。

tasklet 属于 TASKLET_SOFTIRQ 软件中断，入口函数为 tasklet_action，这在内核 kernel\softirq.c 中设置：

```
void __init softirq_init(void)
{
    int cpu;

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

    open_softirq(TASKLET_SOFTIRQ, tasklet_action);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}
```

当驱动程序调用 tasklet_schedule 时，会设置 tasklet 的 state 为 TASKLET_STATE_SCHED，并把它放入某个链表：

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state)) // 1. 如果未设置为SCHED
        __tasklet_schedule(t);                             // 设置为SCHED并放入队列
}

void __tasklet_schedule(struct tasklet_struct *t)
{
    unsigned long flags;

    local_irq_save(flags);   2. 放入队列
    t->next = NULL;
    *__this_cpu_read(tasklet_vec.tail) = t;
    __this_cpu_write(tasklet_vec.tail, &(t->next));
    raise_softirq_irqoff(TASKLET_SOFTIRQ);
    local_irq_restore(flags);   3. 出发TASKLET软中断
}
```

当发生硬件中断时，内核处理完硬件中断后，会处理软件中断。对于 TASKLET_SOFTIRQ 软件中断，会调用 tasklet_action 函数。

执行过程还是挺简单的:从队列中找到 tasklet,进行状态判断后执行 func 函数,从队列中删除 tasklet。

从这里可以看出：

① tasklet_schedule 调度 tasklet 时，其中的函数并不会立刻执行，而只是把 tasklet 放入队列；

② 调用一次 tasklet_schedule，只会导致 tasklnet 的函数被执行一次；

③ 如果 tasklet 的函数尚未执行，多次调用 tasklet_schedule 也是无效的，只会放入队列一次。

tasklet_action 函数解析如下：

```
static __latent_entropy void tasklet_action(struct softirq_action *a)
{
    struct tasklet_struct *list;

    local_irq_disable();
    list = __this_cpu_read(tasklet_vec.head);
    __this_cpu_write(tasklet_vec.head, NULL);
    __this_cpu_write(tasklet_vec.tail, this_cpu_ptr(&tasklet_vec.head));
    local_irq_enable();

    while (list) {
        struct tasklet_struct *t = list;

        list = list->next;

        if (tasklet_trylock(t)) {
            if (!atomic_read(&t->count)) {
                if (!test_and_clear_bit(TASKLET_STATE_SCHED,
                                        &t->state))
                    BUG();
                t->func(t->data); // 3. 执行
                tasklet_unlock(t);
                continue;
            }
            tasklet_unlock(t);
        }

        local_irq_disable();
        t->next = NULL;
        *__this_cpu_read(tasklet_vec.tail) = t;
        __this_cpu_write(tasklet_vec.tail, &(t->next));
        __raise_softirq_irqoff(TASKLET_SOFTIRQ);
        local_irq_enable();
    } « end while list »
} « end tasklet_action »
```

1. 从列表中去出每一项

2. 判断
如果不是SCHED状态，
就是有BUG

4. 从队列中取出

19.7 工作队列

前面讲的定时器、下半部 tasklet，它们都是在中断上下文中执行，它们无法休眠。当要处理更复杂的事情时，往往更耗时。这些更耗时的工作放在定时器或是下半部中，会使得系统很卡；并且循环等待某件事情完成也太浪费 CPU 资源了。

如果使用线程来处理这些耗时的工作，那就可以解决系统卡顿的问题：因为线程可以休眠。

在内核中，我们并不需要自己去创建线程，可以使用“工作队列”(workqueue)。内核初始化工作队列是，就为它创建了内核线程。以后我们要使用“工作队列”，只需要把“工作”放入“工作队列中”，对应的内核线程就会取出“工作”，执行里面的函数。

在 2.xx 的内核中，工作队列的内部机制比较简单；在现在 4.x 的内核中，工作队列的内部机制做得复杂无比，但是用法是一样的。

工作队列的应用场合：要做的事情比较耗时，甚至可能需要休眠，那么可以使用工作队列。

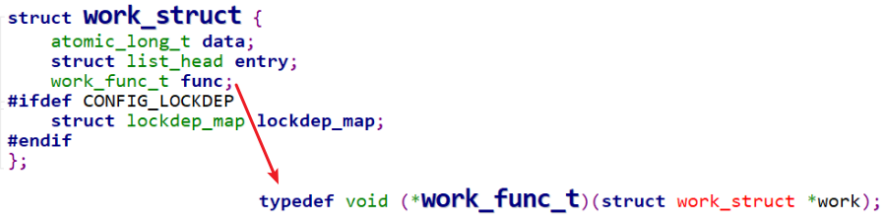
缺点：多个工作(函数)是在某个内核线程中依序执行的，前面函数执行很慢，就会影响到后面的函数。

在多 CPU 的系统下，一个工作队列可以有多个内核线程，可以在一定程度上缓解这个问题。

我们先使用看看怎么使用工作队列。

19.7.1 内核函数

内核线程、工作队列(workqueue)都由内核创建了，我们只是使用。使用的核心是一个 work_struct 结构体，定义如下：



使用工作队列时，步骤如下：

① 构造一个 work_struct 结构体，里面有函数；

② 把这个 work_struct 结构体放入工作队列，内核线程就会运行 work 中的函数。

1. 定义 work

参考内核头文件：include\linux\workqueue.h

```
#define DECLARE_WORK(n, f)                      \
    struct work_struct n = __WORK_INITIALIZER(n, f)

#define DECLARE_DELAYED_WORK(n, f)                  \
    struct delayed_work n = __DELAYED_WORK_INITIALIZER(n, f, 0)
```

第 1 个宏是用来定义一个 work_struct 结构体，要指定它的函数。

第 2 个宏用来定义一个 delayed_work 结构体，也要指定它的函数。所以“delayed”，意思就是说要让它运行时，可以指定：某段时间之后你再执行。

如果要在代码中初始化 work_struct 结构体，可以使用下面的宏：

```
#define INIT_WORK(_work, _func)
```

2. 使用 work：schedule_work

调用 schedule_work 时，就会把 work_struct 结构体放入队列中，并唤醒对应的内核线程。内核线程就会从队列里把 work_struct 结构体取出来，执行里面的函数。

3. 其他函数

序号	函数	说明
1	create_workqueue	在 Linux 系统中已经有了现成的 system_wq 等工作队列， 你当然也可以自己调用 create_workqueue 创建工作队列， 对于 SMP 系统，这个工作队列会有多个内核线程与它对应， 创建工作队列时，内核会帮这个工作队列创建多个内核线程
2	create_singlethread_workqueue	如果想只有一个内核线程与工作队列对应， 可以用本函数创建工作队列， 创建工作队列时，内核会帮这个工作队列创建一个内核线程
3	destroy_workqueue	销毁工作队列
4	schedule_work	调度执行一个具体的 work， 执行的 work 将会被挂入 Linux 系统提供的工作队列
5	schedule_delayed_work	延迟一定时间去执行一个具体的任务， 功能与 schedule_work 类似，多了一个延迟时间
6	queue_work	跟 schedule_work 类似， schedule_work 是在系统默认的工作队列上执行一个 work， queue_work 需要自己指定工作队列
7	queue_delayed_work	跟 schedule_delayed_work 类似， schedule_delayed_work 是在系统默认的工作队列上执行一个 work， queue_delayed_work 需要自己指定工作队列
8	flush_work	等待一个 work 执行完毕， 如果这个 work 已经被放入队列，那么本函数等它执行完毕，并且返回 true； 如果这个 work 已经执行完华才调用本函数，那么直接返回 false
9	flush_delayed_work	等待一个 delayed_work 执行完毕， 如果这个 delayed_work 已经被放入队列，那么本函数等它执行完毕，并且返回 true； 如果这个 delayed_work 已经执行完华才调用本函数，那么直接返回 false

19.7.2 编程、上机

19.7.3 内部机制

初学者知道 work_struct 中的函数是运行于内核线程的上下文，这就足够了。

在 2.xx 版本的 Linux 内核中，创建 workqueue 时就会同时创建内核线程；

在 4.xx 版本的 Linux 内核中，内核线程和 workqueue 是分开创建的，比较复杂。

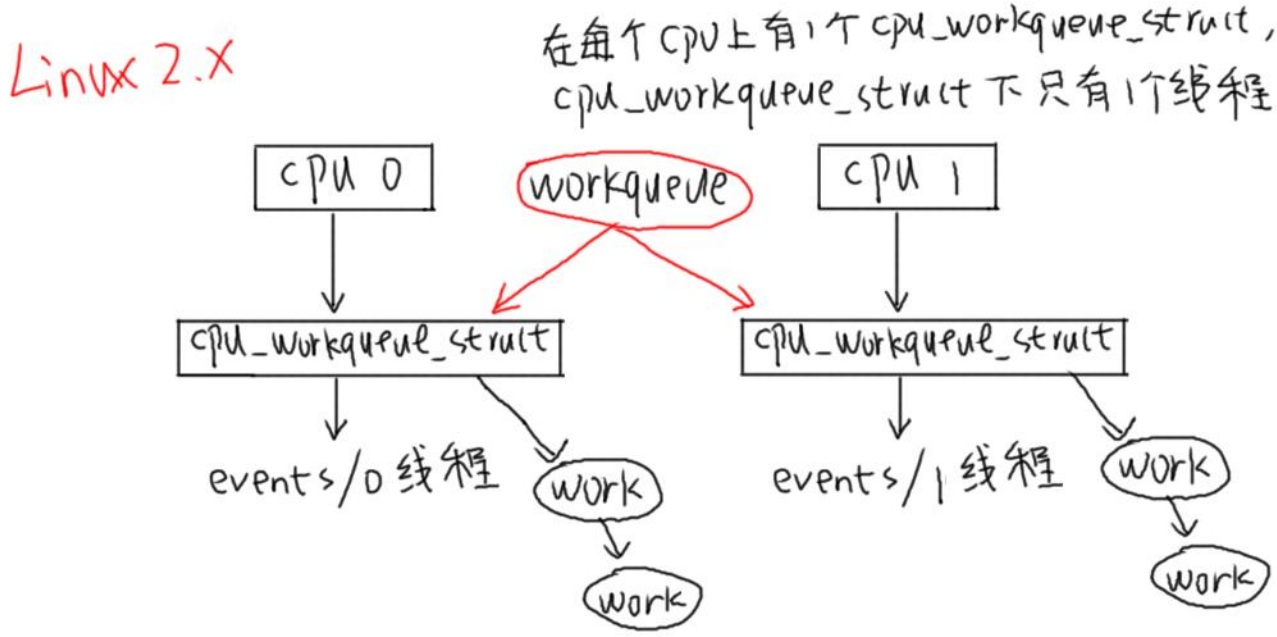
1. Linux 2.x 的工作队列创建过程

代码在 kernel\workqueue.c 中：

```
init_workqueues
    keventd_wq = create_workqueue("events");
        __create_workqueue((name), 0, 0)
            for_each_possible_cpu(cpu) {
                err = create_workqueue_thread(cwq, cpu);
                        p = kthread_create(worker_thread, cwq, fmt, wq->name, cpu);
```

对于每一个 CPU，都创建一个名为“events/X”的内核线程，X 从 0 开始。

在创建 workqueue 的同时创建内核线程。



2. Linux 4.x 的工作队列创建过程

Linux4.x 中，内核线程和工作队列是分开创建的。

先创建内核线程，代码在 kernel\workqueue.c 中：

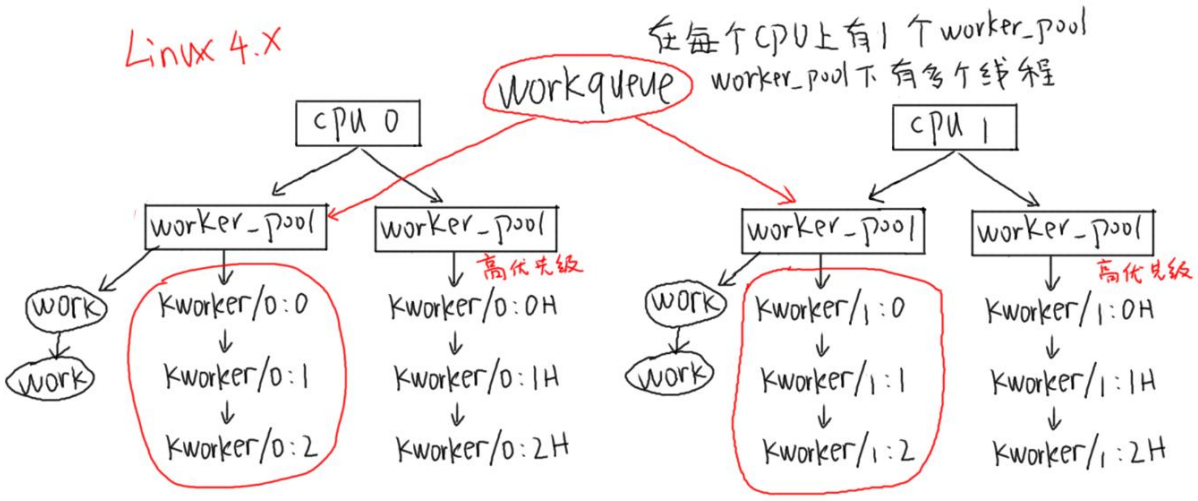
```
init_workqueues
    /* initialize CPU pools */
    for_each_possible_cpu(cpu) {
        for_each_cpu_worker_pool(pool, cpu) {
            /* 对每一个 CPU 都创建 2 个 worker_pool 结构体，它是含有 ID 的 */
            /*  一个 worker_pool 对应普通优先级的 work，第 2 个对应高优先级的 work */
    }

    /* create the initial worker */
    for_each_online_cpu(cpu) {
        for_each_cpu_worker_pool(pool, cpu) {
            /* 对每一个 CPU 的每一个 worker_pool，创建一个 worker */
            /* 每一个 worker 对应一个内核线程 */
            BUG_ON(!create_worker(pool));
        }
    }
```

create_worker 函数代码如下：

```
static struct worker *create_worker(struct worker_pool *pool)
{
    struct worker *worker = NULL;
    int id = -1;
    char id_buf[16];

    /* ID is needed to determine kthread name */
    id = ida_simple_get(&pool->worker_ida, 0, 0, GFP_KERNEL);
    if (id < 0)
        goto fail;

    worker = alloc_worker(pool->node);
    if (!worker)
        goto fail;

    worker->pool = pool;
    worker->id = id;
                          在哪个CPU上运行        poll中第几个线程
    if (pool->cpu >= 0)
        snprintf(id_buf, sizeof(id_buf), "%d:%d%s", pool->cpu, id,
             pool->attrs->nice < 0  ? "H" : "");  H: 高优先级
    else
        snprintf(id_buf, sizeof(id_buf), "u%d:%d", pool->id, id);

    worker->task = kthread_create_on_node(worker_thread, worker, pool->node,
                          "kworker/%s", id_buf);内核线程的名字
```

创建好内核线程后，再创建 workqueue，代码在 kernel\workqueue.c 中：

```
init_workqueues
    system_wq = alloc_workqueue("events", 0, 0);
        __alloc_workqueue_key
            wq = kzalloc(sizeof(*wq) + tbl_size, GFP_KERNEL);  // 分配 workqueue_struct
            alloc_and_link_pwqs(wq) // 跟 worker_poll 建立联系
```



一开始时，每一个 worker_poll 下只有一个线程，但是系统会根据任务繁重程度动态创建、销毁内核线程。所以你可以在 work 中打印线程 ID，发现它可能是变化的。

参考文章：

https://zhuanlan.zhihu.com/p/91106844

https://www.cnblogs.com/vedic/p/11069249.html

https://www.cnblogs.com/zxc2man/p/4678075.html

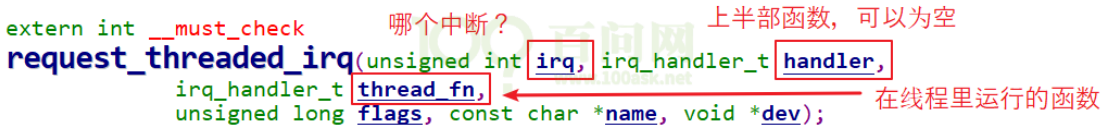
19.8 中断的线程化处理

请先回顾《18.2.7 新技术：threaded irq》。

复杂、耗时的事情，尽量使用内核线程来处理。上节视频介绍的工作队列用起来挺简单，但是它有一个缺点：工作队列中有多个 work，前一个 work 没处理完会影响后面的 work。解决方法有很多种，比如干脆自己创建一个内核线程，不跟别的 work 凑在一块了。在 Linux 系统中，对于存储设备比如 SD/TF 卡，它的驱动程序就是这样做的，它有自己的内核线程。

对于中断处理，还有另一种方法：threaded irq，线程化的中断处理。中断的处理仍然可以认为分为上半部、下半部。上半部用来处理紧急的事情，下半部用一个内核线程来处理，这个内核线程专用于这个中断。

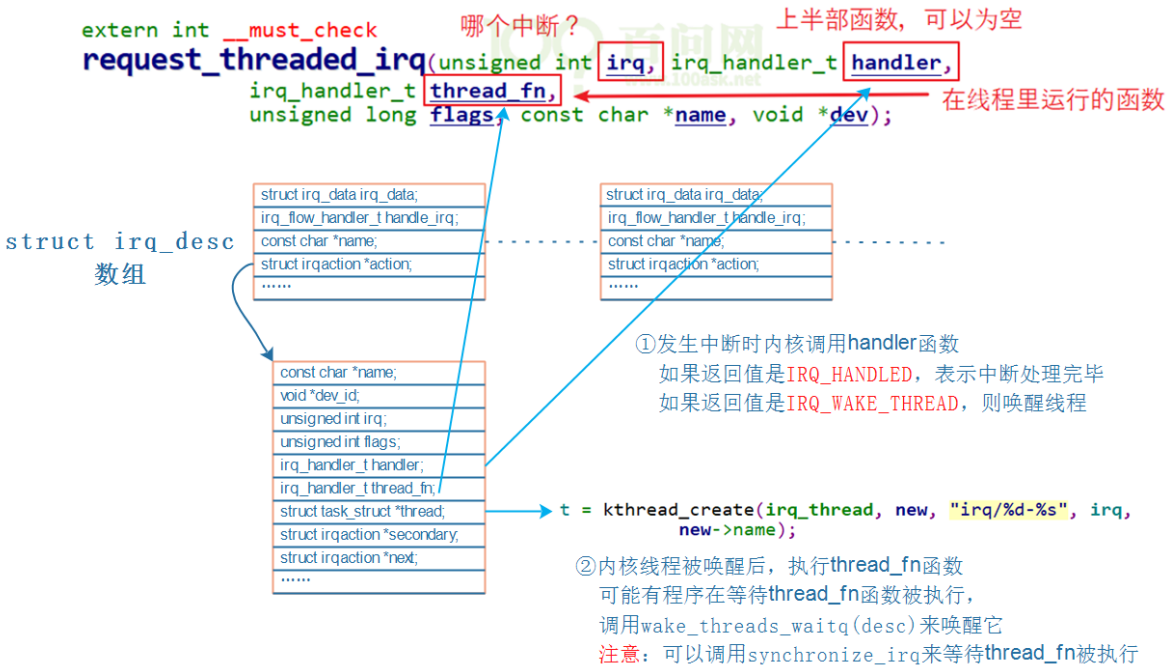
内核提供了这个函数：



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，系统会立刻调用 handler 函数，然后唤醒某个内核线程，内核线程再来执行 thread_fn 函数。

19.8.1 内核机制

1. 调用 request_threaded_irq 后内核的数据结构



2. request_threaded_irq

request_threaded_irq 函数，肯定会创建一个内核线程。

源码在内核文件 kernel\irq\manage.c 中，

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
             irq_handler_t thread_fn, unsigned long irqflags,
             const char *devname, void *dev_id)
{
    // 分配、设置一个 irqaction 结构体
    action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
    if (!action)
        return -ENOMEM;

    action->handler = handler;
    action->thread_fn = thread_fn;
    action->flags = irqflags;
    action->name = devname;
    action->dev_id = dev_id;

    retval = __setup_irq(irq, desc, action);  // 进一步处理
}
```

__setup_irq 函数代码如下(只摘取重要部分)：

```
    if (new->thread_fn && !nested) {
        ret = setup_irq_thread(new, irq, false);
```

setup_irq_thread 函数代码如下(只摘取重要部分)：

```
    if (!secondary) {
        t = kthread_create(irq_thread, new, "irq/%d-%s", irq,
                   new->name);
    } else {
        t = kthread_create(irq_thread, new, "irq/%d-s-%s", irq,
                   new->name);
        param.sched_priority -= 1;
    }
    new->thread = t;
```

3. 中断的执行过程

对于 GPIO 中断，我使用 QEMU 的调试功能找出了所涉及的函数调用，其他板子可能稍有不同。

调用关系如下，反过来看：

```
Breakpoint 1, gpio_keys_gpio_isr (irq=200, dev_id=0x863e6930) at drivers/input/keyboard/gpio_keys.c:393
393 {
(gdb) bt
#0  gpio_keys_gpio_isr (irq=200, dev_id=0x863e6930) at drivers/input/keyboard/gpio_keys.c:393
#1  0x80270528 in __handle_irq_event_percpu (desc=0x8616e300, flags=0x86517edc) at kernel/irq/handle.c:145
#2  0x802705cc in handle_irq_event_percpu (desc=0x8616e300) at kernel/irq/handle.c:185
#3  0x80270640 in handle_irq_event (desc=0x8616e300) at kernel/irq/handle.c:202
#4  0x802738e8 in handle_level_irq (desc=0x8616e300) at kernel/irq/chip.c:518
#5  0x8026f7f8 in generic_handle_irq_desc (desc=<optimized out>) at ./include/linux/irqdesc.h:150
#6  generic_handle_irq (irq=<optimized out>) at kernel/irq/irqdesc.c:590
#7  0x805005e0 in mxc_gpio_irq_handler (port=0xc8, irq_stat=2252237104) at drivers/gpio/gpio-mxc.c:274
#8  0x805006fc in mx3_gpio_irq_handler (desc=<optimized out>) at drivers/gpio/gpio-mxc.c:291
#9  0x8026f7f8 in generic_handle_irq_desc (desc=<optimized out>) at ./include/linux/irqdesc.h:150
#10 generic_handle_irq (irq=<optimized out>) at kernel/irq/irqdesc.c:590
#11 0x8026fd0c in __handle_domain_irq (domain=0x86006000, hwirq=32, lookup=true, regs=0x86517fb0) at kernel/irq/irqdesc.c:627
#12 0x80201484 in handle_domain_irq (regs=<optimized out>, hwirq=<optimized out>, domain=<optimized out>) at ./include/linux/irqdesc.h:168
#13 gic_handle_irq (regs=0xc8) at drivers/irqchip/irq-gic.c:364
#14 0x8020b704 in __irq_usr () at arch/arm/kernel/entry-armv.S:464
```

我们只需要分析__handle_irq_event_percpu 函数，它在 kernel\irq\handle.c 中：

```
irqreturn_t __handle_irq_event_percpu(struct irq_desc *desc, unsigned int *flags)
{
    irqreturn_t retval = IRQ_NONE;
    unsigned int irq = desc->irq_data.irq;
    struct irqaction *action;

    for_each_action_of_desc(desc, action) {
        irqreturn_t res;

        trace_irq_handler_entry(irq, action);
        res = action->handler(irq, action->dev_id);   1.调用上半部处理函数
        trace_irq_handler_exit(irq, action, res);

        if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pF enabled interrupts\n",
                      irq, action->handler))
            local_irq_disable();

        switch (res) {
        case IRQ_WAKE_THREAD:   2. 如果返回值是IRQ_WAKE_THREAD
            /*
             * Catch drivers which return WAKE_THREAD but
             * did not set up a thread function
             */
            if (unlikely(!action->thread_fn)) {
                warn_no_thread(irq, action);
                break;
            }

            __irq_wake_thread(desc, action);   就唤醒中断对应的内核线程
```

线程的处理函数为 irq_thread，代码在 kernel\irq\handle.c 中：

```
while (!irq_wait_for_interrupt(action)) { // 1.休眠等待中断
    irqreturn_t action_ret;

    irq_thread_check_affinity(desc, action);

    action_ret = handler_fn(desc, action); // 2.执行thread_fn
    if (action_ret == IRQ_HANDLED)
        atomic_inc(&desc->threads_handled);
    if (action_ret == IRQ_WAKE_THREAD)
        irq_wake_secondary(desc, action);

    wake_threads_waitq(desc); // 3. 唤醒等待thread_fn的线程
}
```

19.8.2 编程、上机

调用 request_threaded_irq 函数注册中断，调用 free_irq 卸载中断。

从前面可知，我们可以提供上半部函数，也可以不提供：

① 如果不提供

内核会提供默认的上半部处理函数：irq_default_primary_handler，它是直接返回 IRQ_WAKE_THREAD。

② 如果提供的话

返回值必须是：IRQ_WAKE_THREAD。

在 thread_fn 中，如果中断被正确处理了，应该返回 IRQ_HANDLED。

19.9 同步机制

19.10 mmap