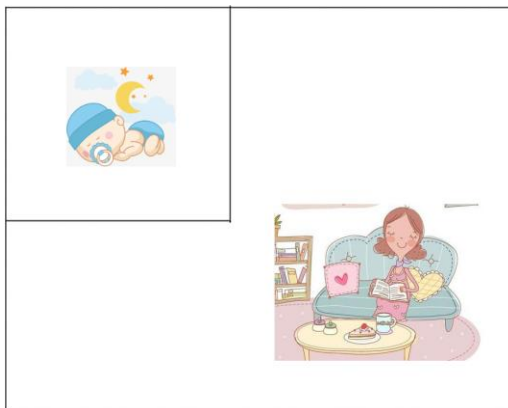


第 19 章 驱动程序基石

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：



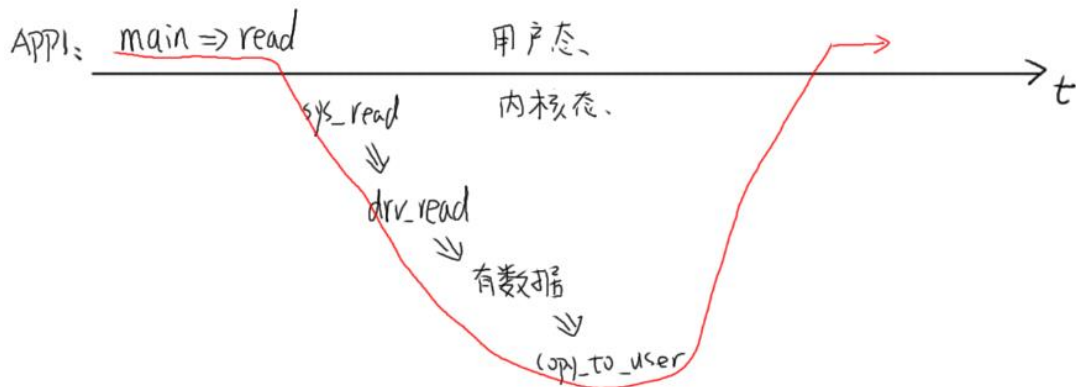
妈妈怎么知道卧室里小孩醒了？

- 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

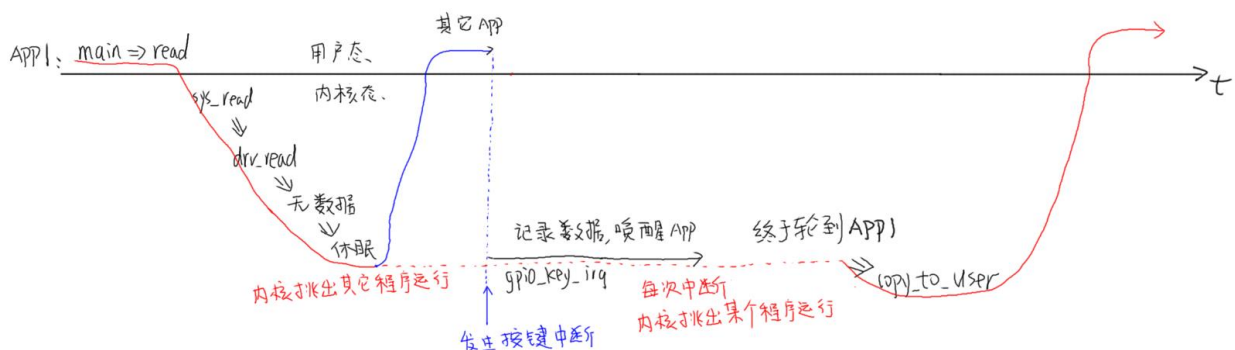
当应用程序必须等待某个事件发生，比如必须等待按键被按下时，**可以使用“休眠-唤醒”机制：**

- 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”，或者说这样：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”。

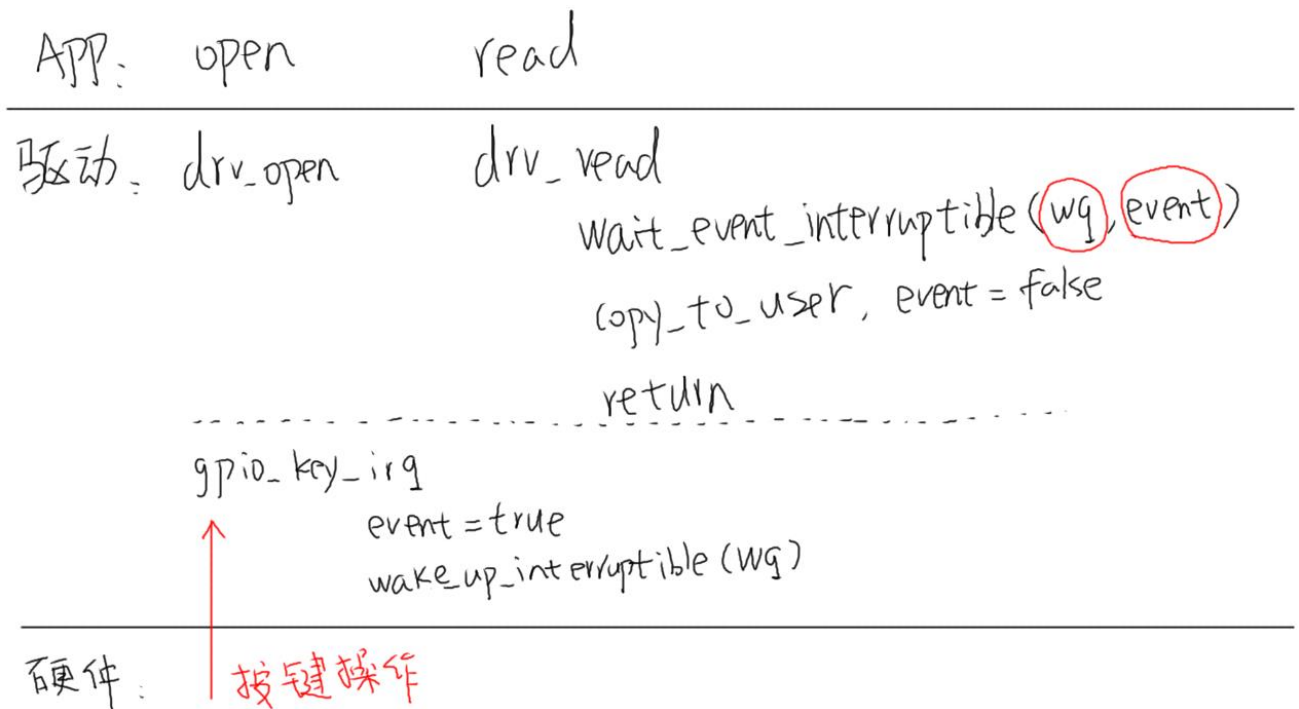
19.1.2.2 唤醒函数

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` 队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 `FALSE` 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，可以在一定程度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);
```

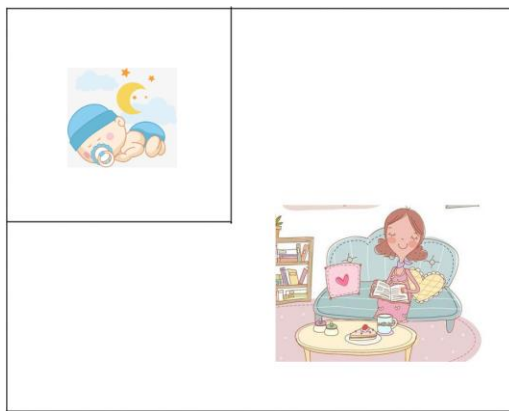
19.2 POLL 机制

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

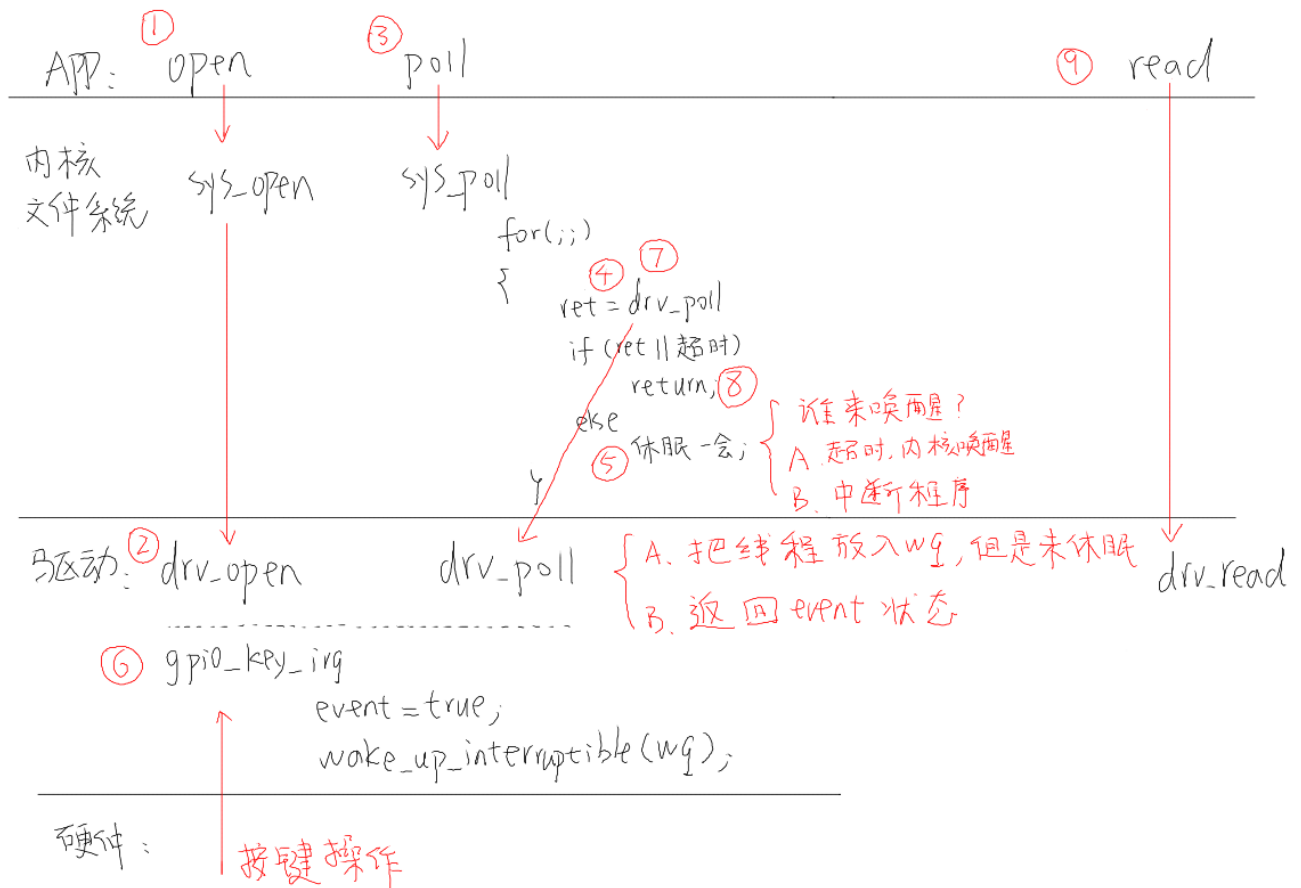
19.2.2 使用流程

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：①~⑧



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，有可能会发生 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：

```

static int do_poll(struct poll_list *list, struct poll_wqueues *wait,
                  struct timespec64 *end_time)
{
    .....
    for (; pfd != pfd_end; pfd++) {
        ① if (!pollfd(pfd, pt, &can_busy_loop,
            // busy_flag)) {
            count++;
            pt->gproc = NULL; //
            /* Round something, stop busy polling */
            busy_flag = 0;
            can_busy_loop = false;
        }

        ⑦ pt->gproc = NULL;

        ⑧ if (count || timed_out)
            break;

        ⑨ if (!poll_schedule_timeout(wait, TASK_INTERRUPTIBLE, to, slack))
            timed_out = 1;
    }
}

mask = 0;
fd = pollfd->fd;
if (fd >= 0) {
    struct fd f = fdget(fd);
    mask = POLLINVAL;
    if (f.file) {
        mask = DEFAULT_POLLMASK;
        if (f.file->f_op->poll) {
            pwait->key = pollfd->events | POLLERR | POLLHUP;
            pwait->key |= busy_flag;
            ② 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->events = mask;
    return mask;
}

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_busy_empty();
}

static inline void poll_wait(struct file
    {
        if (p && p->gproc && wait_address)
            p->gproc(filp, wait_address, p);
    }

/* Add a new entry */
static void __pollwait(struct file *filp, wait_queue
    poll_table *p)
{
    struct poll_wqueues *pwq = container_of(p, struct p
    struct poll_table_entry *entry = poll_get_entry(pwq
    if (!entry)
        return;
    entry->filp = get_file(filp);
    entry->wait_address = wait_address;
    entry->key = p->key;
    init_waitqueue_func_entry(&entry->wait, pollwake);
    entry->wait.private = pwq;
    ⑤ add_wait_queue(wait_address, &entry->wait);
}

```

下次调用poll_wait, 不会再次放入队列

把线程放入队列

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) 1. 驱动程序返回的状态
            *can_busy_poll = true;
    }
    /* Mask out unneeded events. */
    mask &= pollfd->events | POLLERR | POLLHUP;
    fdput(f); 2. 是否是APP期待的？
}
pollfd->revents = mask; 3. 写入revents
```

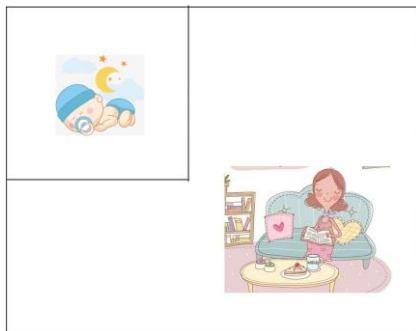
19.3 异步通知

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”。

你付钱后就去看手机了，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
#define SIGPOLL    29
```

← 驱动常用信号
表示有IO事件

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：

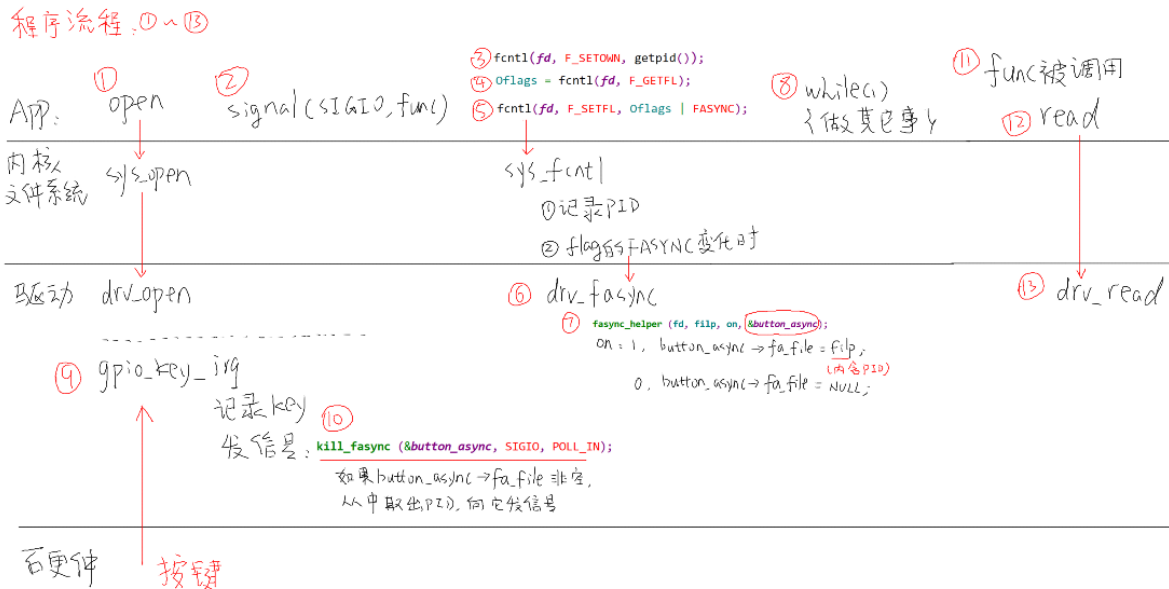
- ① 内核里有那么多驱动，你想让哪一个驱动给你发 SIGIO 信号？
APP 要打开驱动程序的设备节点。
- ② 驱动程序怎么知道要发信号给你而不是别人？
APP 要把自己的进程 ID 告诉驱动程序。
- ③ APP 有时候想收到信号，有时候又不想收到信号：
应该可以把 APP 的意愿告诉驱动。
驱动程序要做什么？发信号。
- ① APP 设置进程 ID 时，驱动程序要记录下进程 ID；
- ② APP 还要使能驱动程序的异步通知功能，驱动中有对应的函数：
APP 打开驱动程序时，内核会创建对应的 `file` 结构体，`file` 中有 `f_flags`；
`f_flags` 中有一个 `FASYNC` 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 fasync_helper，它会根据 FASYNC 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 FASYNC 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_flags` 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

19.6 中断下半部

19.7 工作队列

19.8 中断的线程化处理

19.9 同步机制

19.10 mmap

