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；

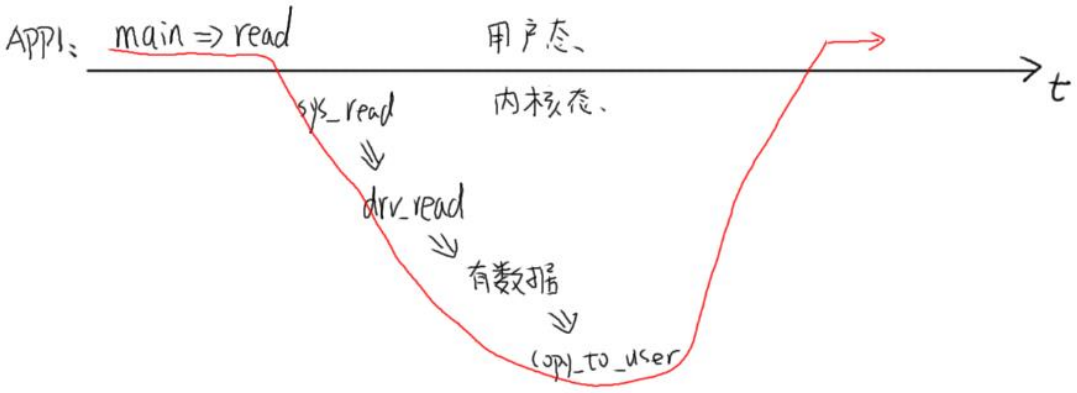
② APP 进入内核态，也就是调用驱动中的对应函数，发现有数据则复制到用户空间并马上返回；

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；

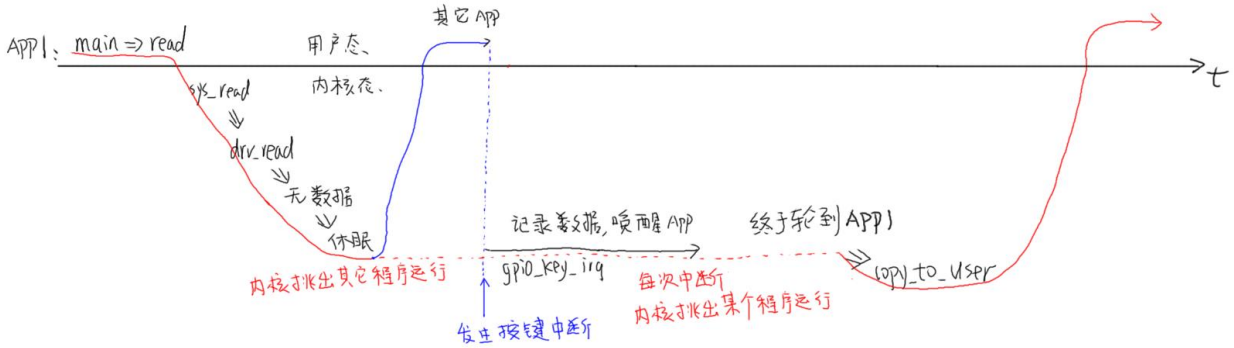
④ 当有数据时，比如当按下按键时，驱动程序的中断服务程序被调用，它会记录数据、唤醒 APP；

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”。

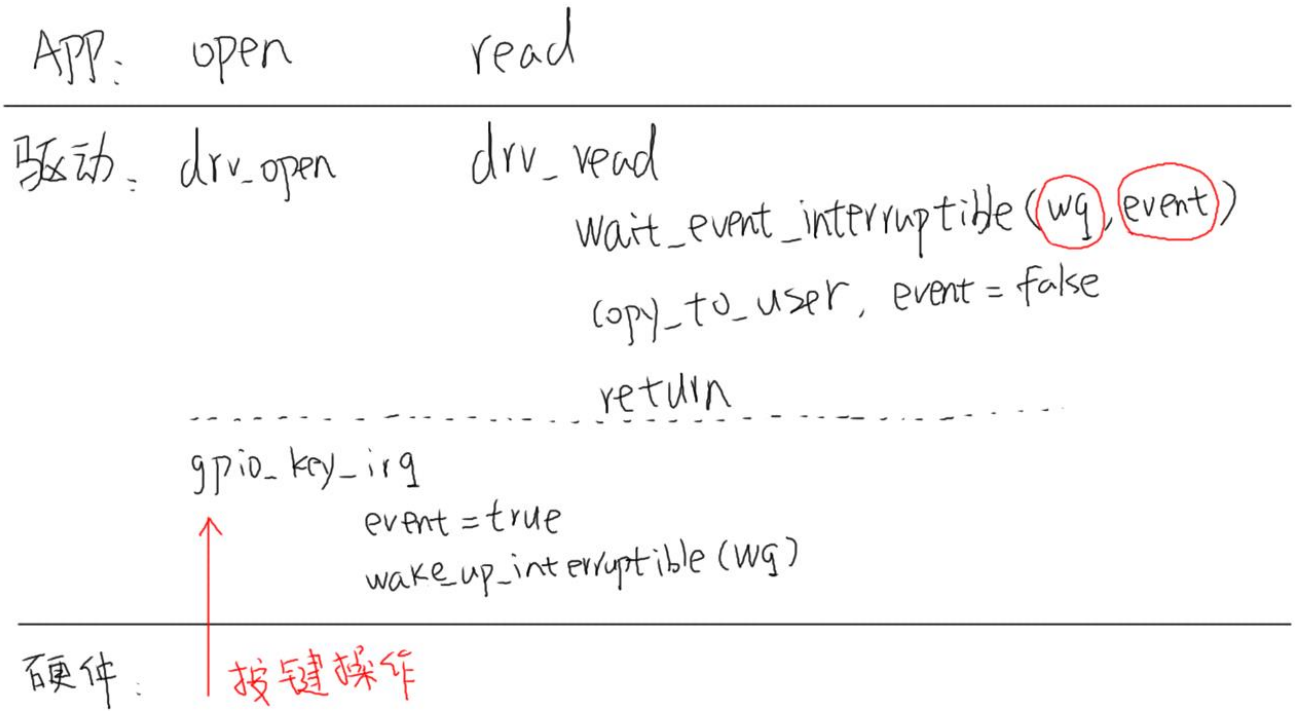
19.1.2.2 唤醒函数

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；

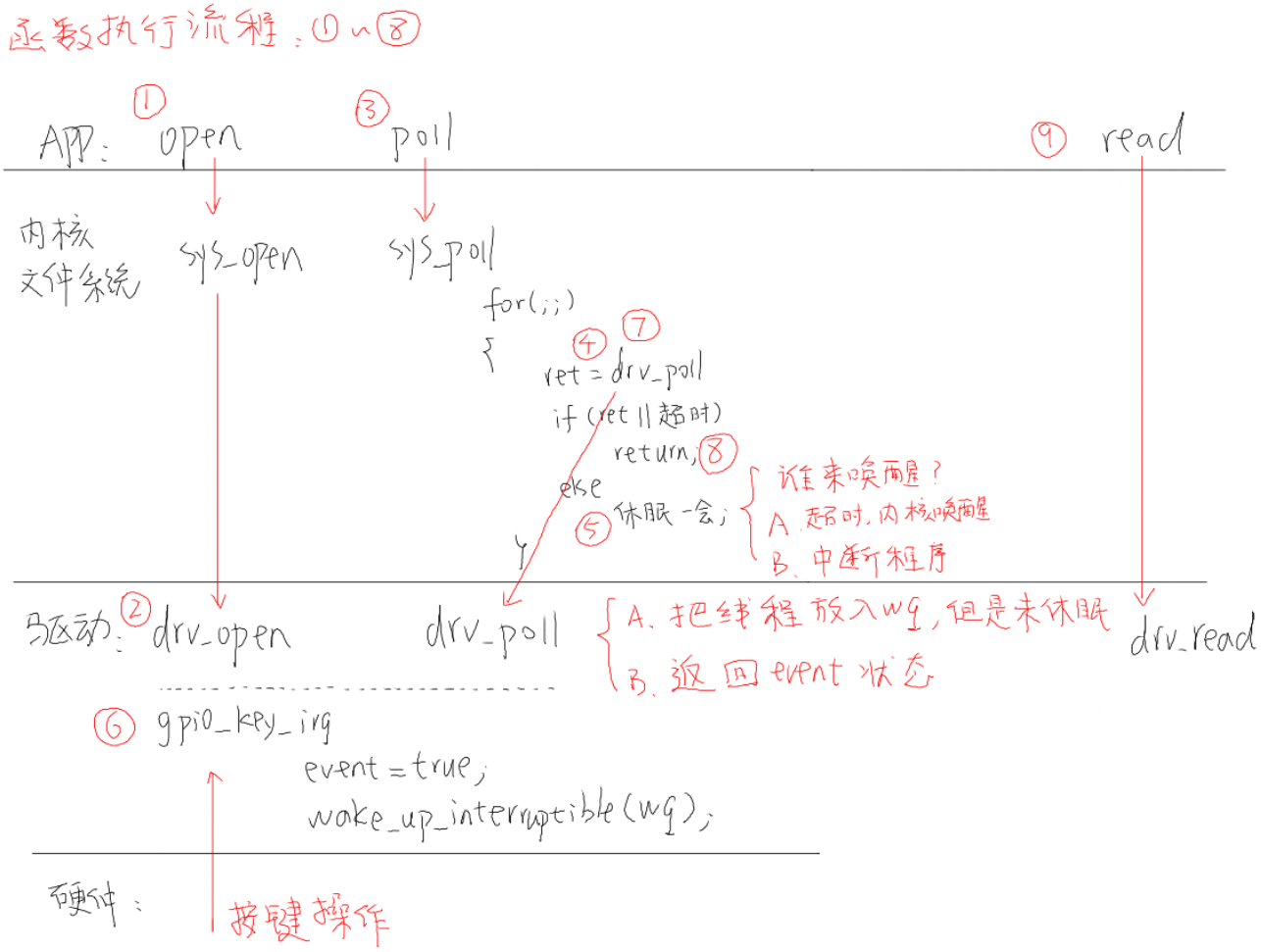
⑥ APP 根据 poll 函数的返回值就可以知道是否有数据，如果有数据就调用 read 得到数据

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：

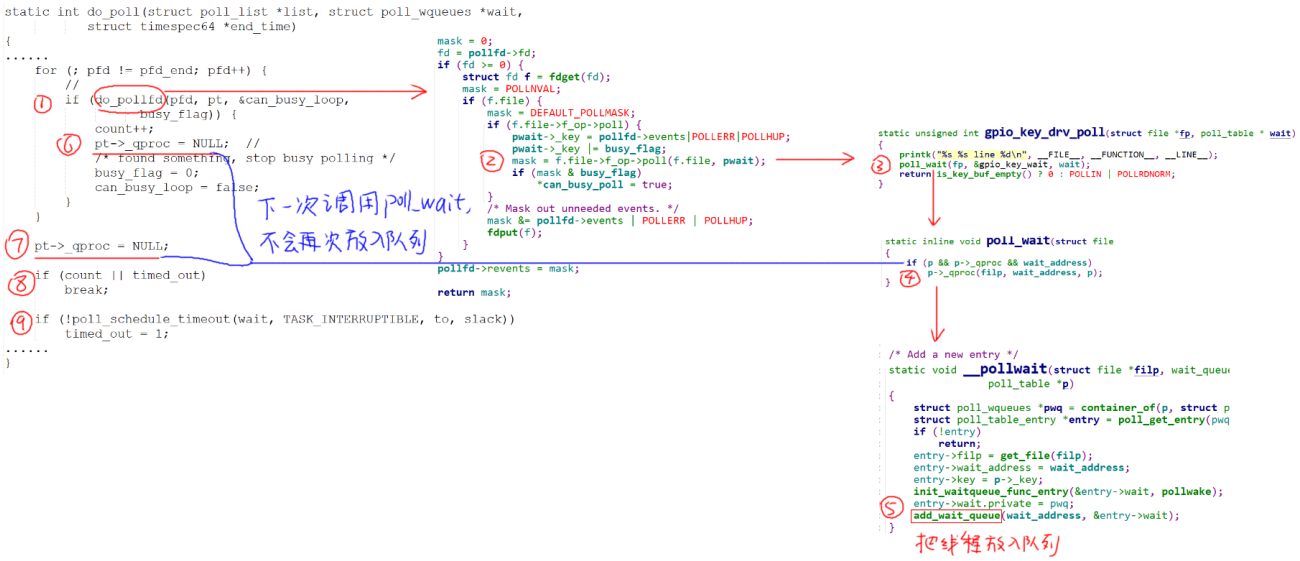
```
poll_initwait
    init_poll_funcptr(&pwq->pt, __pollwait);
        pt->qproc = qproc;
```

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

19.4 定时器

19.5 中断下半部

19.6 工作队列

19.7 中断的线程化处理

19.8 同步机制

19.9 mmap