# 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；

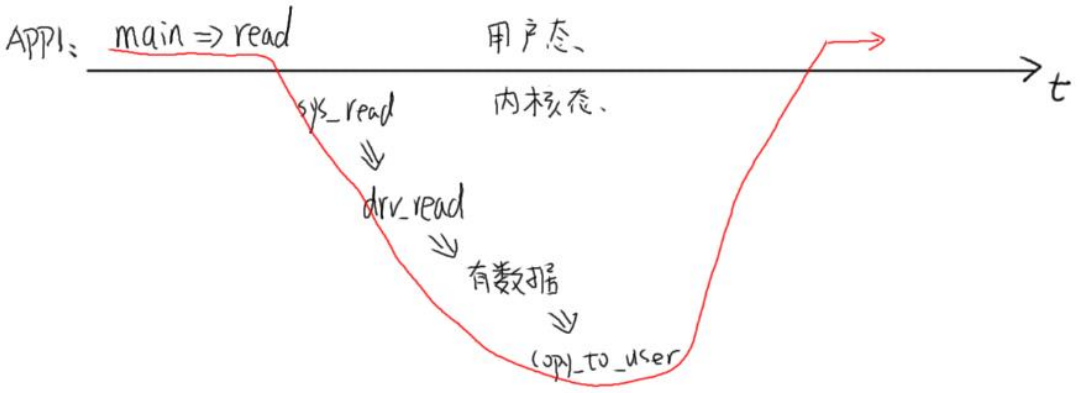
② APP 进入内核态，也就是调用驱动中的对应函数，发现有数据则复制到用户空间并马上返回；

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；

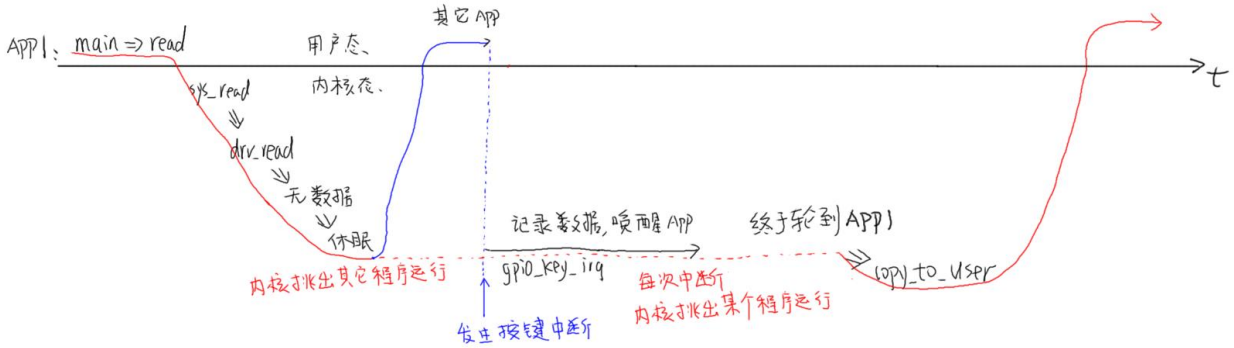
④ 当有数据时，比如当按下按键时，驱动程序的中断服务程序被调用，它会记录数据、唤醒 APP；

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；<br>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；<br>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；<br>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；<br>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”。

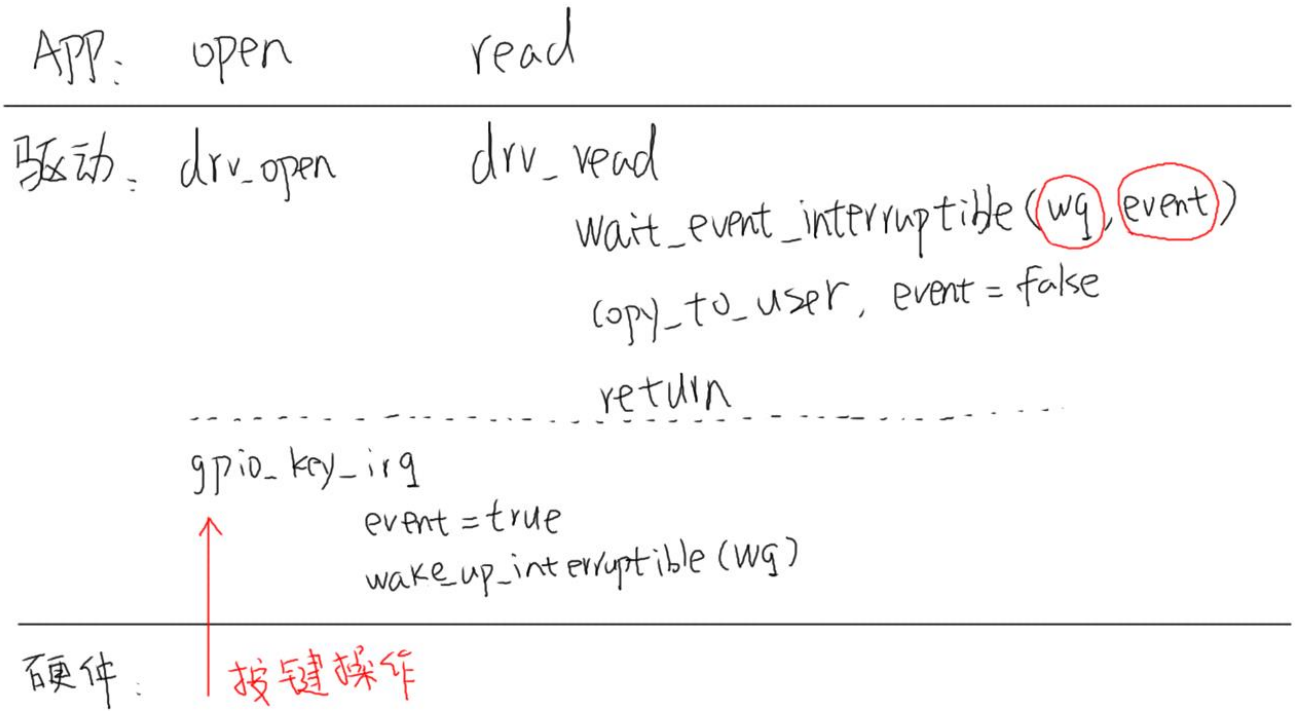
### 19.1.2.2 唤醒函数

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

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