5 并发的处理

Linux 这样的多任务操作系统，都无法避免并发的问题。这一章我们简单了解并发和并发的处理方式。

5.1linux 中的并发

所谓并发就是指多个任务同时访问同一个资源，内存就是资源。内存出错，很有可能导致系统崩溃，所以必须要极力避免并发问题的发生。

并发访问产生的原因众多，比如多线程访问、抢占、中断等等，这些都是系统机制很难避免。要避免并发访问带来的问题，不如去识别出会被同时访问的**共享资源**以及会访问共享资源的程序即**临界区**，并做好保护。当然很多时候识别出需要保护的资源并非易事，这我们只能从不断的积累中去总结。

5.3Linux 对并发的处理

识别出共享资源后，我们就可以使用 Linux 内核提供的保护共享资源机制来避免并发访问。这节我们介绍几种内核提供的并发处理方法。

5.3.1 原子操作

原子操作是指不能被分割或打断的操作，Linux 内核提供了原子变量和系列函数来实现原子操作。文件 include/linux/types.h 中，原子变量定义如下：

```
1.  typedef struct {
2.      int counter;
3.  } atomic_t;
```

定义原子变量的方式为：

```
1.  atomic_t a = ATOMIC_INIT(x);
```

ATOMIC_INIT(x)是用于原子变量初始化的宏定义，x 是我们赋予的初值，不需要的初值的话可以省略。

原子变量的读写操作也要内核提供的接口函数，除了上面说的 ATOMIC_INIT(x)外，还有些常用的接口函数如下：

int atomic_read(atomic_t *v) ： 读取 v 并且返回。

void atomic_set(atomic_t *v, int i) ： 把 i 写入 v。

void atomic_add(int i, atomic_t *v) ： v 的值减去 i。

void atomic_sub(int i, atomic_t *v) ： v 的值加上 i。

void atomic_inc(atomic_t *v) ： v 自增。

void atomic_dec(atomic_t *v) ： v 自减。

int atomic_inc_return(atomic_t *v) ： v 自增并返回。

int atomic_dec_return(atomic_t *v) ： v 自减并返回。

当然还不止这些，除了对整型的操作，内核还提供了原子的位操作，等需要的时候再去了解吧。

具体用法我们放到之后的实验中再去分析。

5.3.2 锁机制

原子操作只能对整形变量和位起到保护作用，很多场景都不适用。比如需要保护结构多变的结构体变量时就需要用到其他的机制，比如这节要讲的锁机制。内核中的锁也用很多种，先介绍一种常用的自旋锁。

自旋锁的机制是，某个线程要访问共享资源时，需要先获取相应的锁即上锁，只要不释放这个锁即解锁，别的线程就无法获取锁也就无法访问共享资源。此时需要却没有获取到自旋锁的线程就会一直处于等带状态(阻塞)。线程的等待状态会浪费很多处理器资源，所以占用自旋锁的临界区要尽量的轻便，更不能调用或存在阻塞的函数或逻辑。

所是一个形象比喻，说到底，也就是个结构体，定义如下：

```
1.  typedef struct spinlock {
2.      union {
3.          struct raw_spinlock rlock;
4.  #ifdef CONFIG_DEBUG_LOCK_ALLOC
5.  # define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
6.          struct {
7.              u8 __padding[LOCK_PADSIZE];
8.              struct lockdep_map dep_map;
9.          };
10. #endif
11.     };
12. } spinlock_t;
```

同样的有了数据结构，还得有相应的操作函数，常用的自旋锁接口函数有：

int spin_lock_init(spinlock_t *lock) ： 自旋锁初始化。

void spin_lock(spinlock_t *lock) ： 获取自旋锁。

void spin_unlock(spinlock_t *lock) ： 释放自旋锁。

int spin_trylock(spinlock_t *lock) ： 获取自旋锁，没有获取到则返回 0。

int spin_is_locked(spinlock_t *lock) ： 检查自旋锁是否已被获取，返回 0 则已被获取，其他值则未被获取。

void spin_lock_irq(spinlock_t *lock) ： 禁用本地中断，并获取自旋锁。

void spin_unlock_irq(spinlock_t *lock) ： 恢复本地中断，并释放自旋锁。

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags) ： 保存中断状态，禁用本地中断，并获取自旋锁，flags 为中断状态。

void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags) ： 恢复中断至保存的

状态，恢复本地中断，并获释放旋锁。

使用自旋锁，需要注意：

1. 使用自旋锁的临界区必定不能进入休眠。自旋锁被成功获取后，内核会停用抢占机制。假如现在有个临界区 A 获取了自旋锁，然后进入休眠主动放弃了 CPU 使用权，线程 B 开始运行。线程 B 的临界区也想访问共享数据，但是自旋锁已被占用，B 线程的临界区就一直处于等待自旋锁的状态，而此时内核抢占被禁止，临界区 A 无法获得主动权释放锁，从而发生了死锁。

2.临界区内要避免中断抢占。假设现在有线程获取了自旋锁，在释放之前，被中断抢占了，如果中断也需要获取共享数据，也去申请自旋锁，那中断就会进入等待状态，那就大事不妙了，直接死锁。这里要避免终端抢占的办法，就是使用上面介绍的 spin_lock_irq 函数和 spin_lock_irqsave 函数来申请自旋锁，禁用本地中断。

3.临界区要尽量短。原因上面有解释过了。

使用示例：

```
1.  spinlock_t lock;
2.  spin_lock (&gpioled.lock);  //上锁
3.  /* 临界区 */
4.  spin_unlock (&gpioled.lock);  //解锁
```

上锁和解锁中间就是临界区。

5.3.3 信号量

与自旋锁相比，信号量有两个优势：

1.信号量可以让等待信号量的线程进入休眠，减少 CPU 的占用；

2.信号量支持对个线程同时访问共享资源。

信号量结构体定义如下：

```
1.  struct semaphore {
2.      raw_spinlock_t lock;
3.      unsigned int count;
4.      struct list_head wait_list;
5.  };
```

元素 count 即指信号量支持同时访问共享资源的线程数。

常用的信号量接口函数有：

void sema_init(struct semaphore *sem, int val) ：初始化信号量，设置信号值(同时访问个数)为 val。

void down(struct semaphore *sem) ：获取信号量，失败时会进入休眠且不可以被信号打断，不能用于中断。

int down_trylock(struct semaphore *sem) ：获取信号量，不会进入休眠，成功返回 0。

int down_interruptible(struct semaphore *sem) ：获取信号量，失败时会进入休眠但可以被信号打断，不能用于中断，休眠被打断时返非 0 值。

void up(struct semaphore *sem) ：释放信号量。

使用信号量要注意的地方：

1.线程休眠后会切换线程，如果占用信号量的临界区很短，导致频繁切换线程，也会带来大量的开销，因此与自旋锁相反，**信号量不适用于共享资源使用时间很短的场合**；

2.中断不能休眠，因此**中断中不能使用会进入休眠的方式申请信号量**；

3.信号量**允许多个线程同时访问共享资源时(count > 1)，不可用于互斥访问**(互斥访问指一次只能有一个线程访问共享资源)。

使用示例：

```
1.  struct semaphore sem;
2.  sema_init(&sem, 1);
3.  down(&sem);
4.  /* 临界区 */
5.  up(&sem);
```

5.4 实验

5.4.1 原理图

和章节 1.3.1 的内容相同。

5.4.2 设备树

和上一章 4.3.2 相同。

5.4.3 驱动代码

以上一章的驱动代码为例，首先识别出共享资源。led 设备的驱动，那共享资源就是 led 设备了，如果有多个应用程序来调用这个驱动，最终就是 led 设备会被多个应用程序操作。再进一步的说，就是 led 设备节点，即/dev/gpio_leds 这个设备文件，在操作这个设备时，不能让别的应用程序再操作。而临界区就是从占用这个节点开始(即 open 函数打开/dev/gpio_leds 这个设备文件)到释放这个节点为止(close 设备文件)。

这章有三个机制要实验，但重复的代码很多，为了简便，我把三个机制都写在一个代码里。用宏定义开关来隔开。使用 petalinux 新建驱动名为”ax-concled-dev”，在 ax-concled-dev 中输入以下代码：

```
1.  #include <linux/module.h>
2.  #include <linux/kernel.h>
3.  #include <linux/fs.h>
4.  #include <linux/init.h>
5.  #include <linux/ide.h>
6.  #include <linux/types.h>
7.  #include <linux/errno.h>
```

```
8.  #include <linux/cdev.h>
9.  #include <linux/of.h>
10. #include <linux/of_address.h>
11. #include <linux/of_gpio.h>
12. #include <linux/device.h>
13. #include <linux/delay.h>
14. #include <linux/init.h>
15. #include <linux/gpio.h>
16. #include <asm/uaccess.h>
17. #include <asm/mach/map.h>
18. #include <asm/io.h>
19.
20. /* 设备节点名称 */
21. #define DEVICE_NAME       "gpio_leds"
22. /* 设备号个数 */
23. #define DEVID_COUNT       1
24. /* 驱动个数 */
25. #define DRIVE_COUNT       1
26. /* 主设备号 */
27. #define MAJOR1
28. /* 次设备号 */
29. #define MINOR1            0
30. /* LED 点亮时输入的值 */
31. #define ALINX_LED_ON      1
32. /* LED 熄灭时输入的值 */
33. #define ALINX_LED_OFF     0
34.
35. /* 原子变量开关 */
36. #define ATOMIC_T_ON
37. /* 自旋锁开关 */
38. //#define SPINKLOCK_T_ON
39. /* 信号量开关 */
40. //#define SEMAPHORE_ON
41.
42. /* 把驱动代码中会用到的数据打包进设备结构体 */
43. struct alinx_char_dev{
44.     dev_t             devid;          //设备号
45.     struct cdev       cdev;           //字符设备
46.     struct class      *class;         //类
47.     struct device     *device;        //设备
48.     struct device_node *nd;           //设备树的设备节点
49.     int               alinx_led_gpio; //gpio 号
50.
51. #ifdef ATOMIC_T_ON
```

```
52.     atomic_t            lock;               //原子变量
53. #endif
54. 
55. #ifdef SPINKLOCK_T_ON
56.     spinlock_t          lock;               //自旋锁变量
57.     int                 source_status;      //资源占用状态
58. #endif
59. 
60. #ifdef SEMAPHORE_ON
61.     struct semaphore    lock;
62. #endif
63. };
64. /* 声明设备结构体 */
65. static struct alinx_char_dev alinx_char = {
66.     .cdev = {
67.         .owner = THIS_MODULE,
68.     },
69. };
70. 
71. /* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
72. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
73. {
74.     /* 应用程序调用了 open 函数表示需要调用共享资源 */
75. #ifdef ATOMIC_T_ON
76.     /* 通过判断原子变量的值来判断资源的占用状态 */
77.     if (!atomic_read(&alinx_char.lock))
78.     {
79.         /* 若原子变量值为 0，则资源没有被占用，
80.         此时把原子变量加 1，表示之后资源就被占用了 */
81.         atomic_inc(&alinx_char.lock);
82.     }
83.     else
84.     {
85.         /* 否则资源被占用，返回忙碌 */
86.         return -EBUSY;
87.     }
88. #endif
89. 
90. #ifdef SPINKLOCK_T_ON
91.     /* 获取自旋锁 */
92.     spin_lock(&alinx_char.lock);
93.     /* 判断资源占用状态 */
94.     if(!alinx_char.source_status)
95.     {
```

```
96.         /* 为0则未被占用,
97.         此时把状态值加1, 表示之后资源就被占用了 */
98.         alinx_char.source_status ++;
99.         /* 释放锁 */
100.        spin_unlock(&alinx_char.lock);
101.    }
102.    else
103.    {
104.        /* 释放锁 */
105.        spin_unlock(&alinx_char.lock);
106.        /* 否则资源被占用, 返回忙碌 */
107.        return -EBUSY;
108.    }
109. #endif
110.
111. #ifdef SEMAPHORE_ON
112.    /* 获取信号量 */
113.    down(&alinx_char.lock);
114. #endif
115.
116.    /* 设置私有数据 */
117.    file_p->private_data = &alinx_char;
118.    printk("gpio_test module open\n");
119.    return 0;
120. }
121.
122.
123. /* write 函数实现, 对应到 Linux 系统调用函数的 write 函数 */
124. static ssize_t gpio_leds_write(struct file *file_p, const char __user *buf, size_t len, lof
    f_t *loff_t_p)
125. {
126.    int retvalue;
127.    unsigned char databuf[1];
128.    /* 获取私有数据 */
129.    struct alinx_char_dev *dev = file_p->private_data;
130.
131.    retvalue = copy_from_user(databuf, buf, len);
132.    if(retvalue < 0)
133.    {
134.        printk("alinx led write failed\r\n");
135.        return -EFAULT;
136.    }
137.
138.    if(databuf[0] == ALINX_LED_ON)
```

```
139.     {
140.         gpio_set_value(dev->alinx_led_gpio, !!0);
141.     }
142.     else if(databuf[0] == ALINX_LED_OFF)
143.     {
144.         gpio_set_value(dev->alinx_led_gpio, !!1);
145.     }
146.     else
147.     {
148.         printk("gpio_test para err\n");
149.     }
150. 
151.     return 0;
152. }
153. 
154. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
155. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
156. {
157.     /* 应用程序调用 close 函数，宣布资源已使用完毕 */
158. #ifdef ATOMIC_T_ON
159.     /* 原子变量恢复为 0，表示资源已使用完毕 */
160.     atomic_set(&alinx_char.lock, 0);
161. #endif
162. 
163. #ifdef SPINKLOCK_T_ON
164.     /* 获取自旋锁 */
165.     spin_lock(&alinx_char.lock);
166.     /* 资源占用状态恢复为 0，表示资源已使用完毕 */
167.     alinx_char.source_status = 0;
168.     /* 释放锁 */
169.     spin_unlock(&alinx_char.lock);
170. #endif
171. 
172. #ifdef SEMAPHORE_ON
173.     /* 释放信号量 */
174.     up(&alinx_char.lock);
175. #endif
176. 
177.     printk("gpio_test module release\n");
178.     return 0;
179. }
180. 
181. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
182. static struct file_operations ax_char_fops = {
```

```
183.     .owner   = THIS_MODULE,
184.     .open    = gpio_leds_open,
185.     .write   = gpio_leds_write,
186.     .release = gpio_leds_release,
187. };
188.
189. /* 模块加载时会调用的函数 */
190. static int __init gpio_led_init(void)
191. {
192.     /* 用于接受返回值 */
193.     u32 ret = 0;
194.
195. #ifdef ATOMIC_T_ON
196.     /* 设置原子变量为0，即资源为未被占用的状态 */
197.     atomic_set(&alinx_char.lock, 0);
198. #endif
199.
200. #ifdef SPINKLOCK_T_ON
201.     /* 初始化自旋锁 */
202.     spin_lock_init(&alinx_char.lock);
203.     /* 初始化资源占用状态为0，意为资源没有被占用 */
204.     alinx_char.source_status = 0;
205. #endif
206.
207. #ifdef SEMAPHORE_ON
208.     /* 初始化信号量 */
209.     sema_init(alinx_char.lock, 1);
210. #endif
211.
212.     /* 获取设备节点 */
213.     alinx_char.nd = of_find_node_by_path("/alinxled");
214.     if(alinx_char.nd == NULL)
215.     {
216.         printk("alinx_char node not find\r\n");
217.         return -EINVAL;
218.     }
219.     else
220.     {
221.         printk("alinx_char node find\r\n");
222.     }
223.
224.     /* 获取节点中gpio标号 */
225.     alinx_char.alinx_led_gpio = of_get_named_gpio(alinx_char.nd, "alinxled-gpios", 0);
226.     if(alinx_char.alinx_led_gpio < 0)
```

```
227.    {
228.        printk("can not get alinxled-gpios");
229.        return -EINVAL;
230.    }
231.    printk("alinxled-gpio num = %d\r\n", alinx_char.alinx_led_gpio);
232.
233.    /* 申请 gpio 标号对应的引脚 */
234.    ret = gpio_request(alinx_char.alinx_led_gpio, "alinxled");
235.    if(ret != 0)
236.    {
237.        printk("can not request gpio\r\n");
238.    }
239.
240.    /* 把这个 io 设置为输出 */
241.    ret = gpio_direction_output(alinx_char.alinx_led_gpio, 1);
242.    if(ret < 0)
243.    {
244.        printk("can not set gpio\r\n");
245.    }
246.
247.    /* 注册设备号 */
248.    alloc_chrdev_region(&alinx_char.devid, MINOR1, DEVID_COUNT, DEVICE_NAME);
249.
250.    /* 初始化字符设备结构体 */
251.    cdev_init(&alinx_char.cdev, &ax_char_fops);
252.
253.    /* 注册字符设备 */
254.    cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
255.
256.    /* 创建类 */
257.    alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
258.    if(IS_ERR(alinx_char.class))
259.    {
260.        return PTR_ERR(alinx_char.class);
261.    }
262.
263.    /* 创建设备节点 */
264.    alinx_char.device = device_create(alinx_char.class, NULL,
265.                                      alinx_char.devid, NULL,
266.                                      DEVICE_NAME);
267.    if (IS_ERR(alinx_char.device))
268.    {
269.        return PTR_ERR(alinx_char.device);
270.    }
```

```
271.
272.     return 0;
273. }
274.
275. /* 卸载模块 */
276. static void __exit gpio_led_exit(void)
277. {
278.     /* 释放 gpio */
279.     gpio_free(alinx_char.alinx_led_gpio);
280.
281.     /* 注销字符设备 */
282.     cdev_del(&alinx_char.cdev);
283.
284.     /* 注销设备号 */
285.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
286.
287.     /* 删除设备节点 */
288.     device_destroy(alinx_char.class, alinx_char.devid);
289.
290.     /* 删除类 */
291.     class_destroy(alinx_char.class);
292.
293.     printk("gpio_led_dev_exit_ok\n");
294. }
295.
296. /* 标记加载、卸载函数 */
297. module_init(gpio_led_init);
298. module_exit(gpio_led_exit);
299.
300. /* 驱动描述信息 */
301. MODULE_AUTHOR("Alinx");
302. MODULE_ALIAS("gpio_led");
303. MODULE_DESCRIPTION("CONCURRENT driver");
304. MODULE_VERSION("v1.0");
305. MODULE_LICENSE("GPL");
```

与上一章有差异的部分加粗了。改动主要集中在：

1.结构体定义中增加成员；

2.驱动入口函数中，添加初始化操作；

3.open 函数中申请；

4.release 函数中释放。

通过同定义开关分成了三部分，分别对应原子操作、自旋锁、信号量，他们的用法都比较相似。按顺序来分析。

原子操作：

看宏定义 ATOMIC_T_ON 相关的部分，思路是：

a：**52** 行定义一个原子变量，并把它作为共享资源使用状态的标志，0 为未被占用，其他值则已被占用。

b：**197** 行在驱动入口函数中初始化原子变量为 0，即共享资源未被占用。

c：**77** 行在 open 函数中，先判断资源占用状态，如果原子变量不为 0，则已被占用返回 busy。否则资源是空闲的，则使原子变量不为 0，然后 open 函数返回 0，表示调用共享资源设备节点成功。

d：**160** 行在 release 函数中，应用程序释放了共享资源，此时把原子变量置为 0，共享资源恢复为空闲的状态。

实际上这种流程下，共享资源还没有做到完全的安全，因为 77 行的 if 语句不是原子操作，这个语句执行的过程中，还是有可能被抢占或被中断打断导致判断的结果与实际有差别。后面的自旋锁和信号量就没有这种问题。

自旋锁：

再看宏定义 SPINKLOCK_T_ON 相关的部分，如果想编译出自旋锁的版本，别忘了把 **38** 行的注释解开，并把 **36** 行和 **40** 行注释。

前面介绍过自旋锁的临界区要尽量短，正常情况下，从应用程序调用 open 到应用程序调用 close 之间都是临界区。这中间的时间不由驱动来决定，有可能会拉的很长，所有在 open 函数中上锁，在 release 函数中解锁就有临界区很长的风险。可以借助上面原子变量实验中用到的标志 flag 的思想来实现：

a：**56** 行定义一个自旋锁，同时 **57** 定义一个标志 source_status 来记录资源的使用状态，0 为未被占用，其他值则已被占用。

b：**202** 行在驱动入口函数中初始化自旋锁，并把资源状态变量初始化成空闲状态 0。

c：**92** 行是上锁，临界区仅做一个判断资源状态值，如果未被占用则标记为占用然后**解锁**，已被占用则直接**解锁**并返回 busy。这样一来，及时资源被占用了，线程也不会一直等待了。

d：**165** 行 release 函数中，获取锁后，设置资源占用状态为 0，然后就解锁。

信号量：

信号量的用法就很简单了，因为信号量的临界区要长，所以我们放心的在 open 中上锁，在 release 中解锁，就不需要用标志了。当然临界区到底有多长，还是由应用程决定的。

5.4.4 测试代码

测试代码在上一章的基础上稍作修改。新建 QT 工程名为”axledlong_test”，新建 main.c，输入下列代码：

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <unistd.h>
4.  #include <fcntl.h>
5.
6.  int main(int argc, char **argv)
7.  {
8.      int fd;
9.      char buf;
```

```
10.     int count;
11.
12.     if(3 != argc)
13.     {
14.         printf("none para\n");
15.         return -1;
16.     }
17.
18.     fd = open(argv[1], O_RDWR);
19.     if(fd < 0)
20.     {
21.         printf("Can't open file %s\r\n", argv[1]);
22.         return -1;
23.     }
24.
25.     if(!strcmp("on",argv[2]))
26.     {
27.         printf("ps_led1 on\n");
28.         buf = 1;
29.         write(fd, &buf, 1);
30.     }
31.     else if(!strcmp("off",argv[2]))
32.     {
33.         printf("ps_led1 off\n");
34.         buf = 0;
35.         write(fd, &buf, 1);
36.     }
37.     else
38.     {
39.         printf("wrong para\n");
40.         return -2;
41.     }
42.
43.     count = 20;
44.     while(count --)
45.     {
46.         sleep(1);
47.     }
48.
49.     close(fd);
50.     return 0;
51. }
```

在 43 到 47 行添加代码，用 20 秒的 sleep 模拟以下对资源 20 秒的占用。其他和之前一

样。

5.4.5 运行测试

加载驱动的步骤就略过了。如下图：

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ./ax-concled-dev.ko
ax_concled_dev: loading out-of-tree module taints kernel.
alinx_char node find
alinxled-gpio num = 899
root@ax_peta:/mnt#
```

```
root@ax_peta:/mnt/build-axledlong_test-ZYNQ-Debug# cd ..
root@ax_peta:/mnt# cd ./build-axledlong_test-ZYNQ-Debug/
root@ax_peta:/mnt/build-axledlong_test-ZYNQ-Debug#
```

用下列命令来打开测试程序：

```
./axledlong_test /dev/gpio_leds on&

./axledlong_test /dev/gpio_leds off
```

“&”表示在后台运行 app，可以使用 top 命令查看后台正在运行的程序。

```
root@ax_peta:/mnt/build-axledlong_test-ZYNQ-Debug# ./axledlong_test /dev/gpio_leds on&
[1] 1328
gpio_test module open
ps_led1 on
root@ax_peta:/mnt/build-axledlong_test-ZYNQ-Debug# ./axledlong_test /dev/gpio_leds off
Can't open file /dev/gpio_leds
root@ax_peta:/mnt/build-axledlong_test-ZYNQ-Debug# gpio_test module release

[1]+  Done                    ./axledlong_test /dev/gpio_leds on
root@ax_peta:/mnt/build-axledlong_test-ZYNQ-Debug# ./axledlong_test /dev/gpio_leds off
gpio_test module open
ps_led1 off
```

可以看出，先执行 app 打开 led，这个线程就占用了设备节点。再执行 off，app 返回了”Can't open file /dev/gpio_leds”，而不是直接关闭 led。等输出” gpio_test module release”时，说明一开始的 app 线程已经释放了资源，再执行 off，led 就被关闭了。

这是原子变量时的测试状态，其他两个实验的步骤也是一样的，自旋锁的现象与原子变量的实验现象是一致的。信号量则会略有区别，因为信号量的实现中，会休眠去等待资源，因此 off 时不会直接返回错误，而是会等输出” gpio_test module release”后，再自动取熄灭 led。