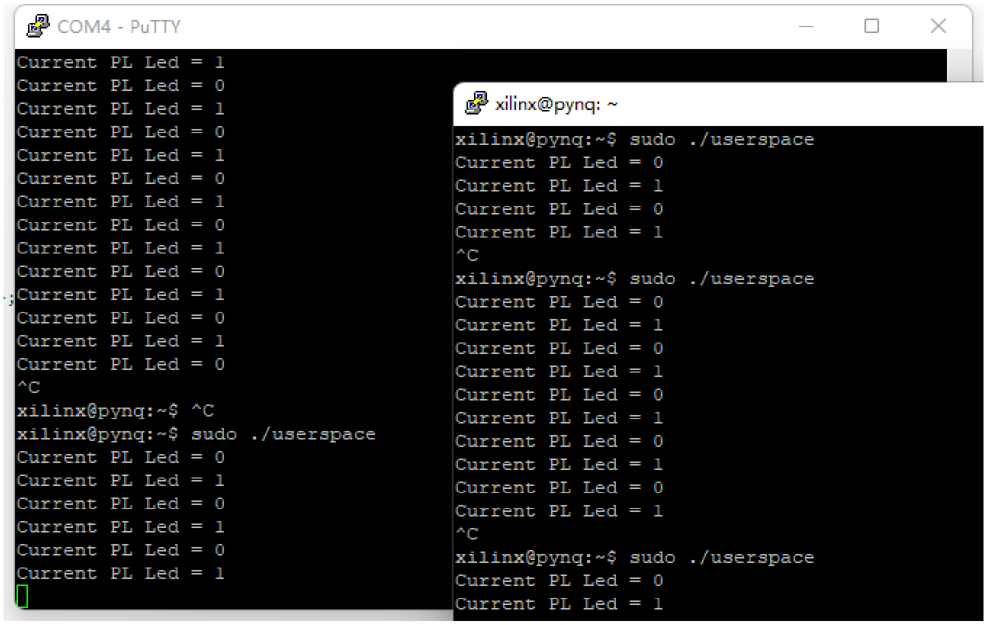
[L21]Linux GPIO 驱动中的并发与竞争

学到这里我们假设你已经掌握FreeRTOS之类的系统,那么接下来的学习如何解决并发竞争问题,首先我们知道,LED只有一个,如果一个程序执行,另一个也来执行,LED资源不就抢着用了,就不是简单的Blink了,不妨我们启动两个用户空间程序试试.



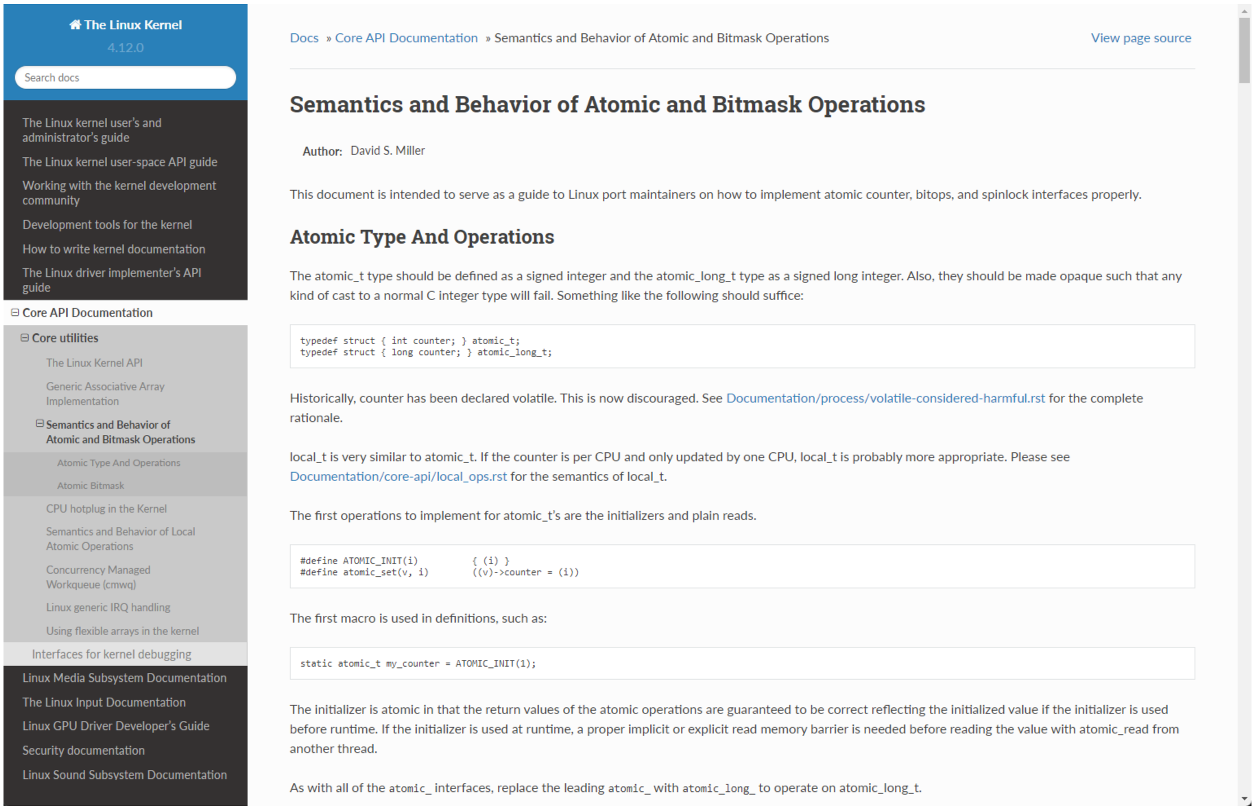
```
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
^C
xilinx@pynq:~$ ^C
xilinx@pynq:~$ sudo ./userspace
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
```

```
xilinx@pynq:~$ sudo ./userspace
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
^C
xilinx@pynq:~$ sudo ./userspace
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
^C
xilinx@pynq:~$ sudo ./userspace
Current PL Led = 0
Current PL Led = 1
```

只要开的越多,LED闪烁就越乱,因为同时有很多个程序正在不断地翻转LED,对于Linux内核,常用有四种保护机制.

1. 原子操作 - 简单的变量加减乘除置位复位操作,对于临界点是片段无效,用户空间无感.
2. 自旋锁 - 拿不到锁要死等浪费资源的,短时间取锁合适,原则上不用于中断上下文(即任何休眠和换出也不可以!),用户空间可立即返回.
3. 信号量 - 拿不到锁就预定一个通知,等到通知来了再干活,适合长时间锁,原则上不用于中断上下文(即任何休眠和换出也不可以!),用户空间可死等,死等过程并不占用CPU.
4. 互斥体 - 只有一把车钥匙,你拿到了别人就拿不到,拿不到的人也不会等,原则上不用于中断上下文(即任何休眠和换出也不可以!).用户空间可死等,死等过程并不占用CPU.

先讲讲原子操作,其所有操作函数都是atomic_开头的,并且变量的定义要用atomic_t修饰,相关的宏定义也是ATOMIC_开头的,具体需要自行查阅文档理解,理解之后继续看我们的目的.


The Linux Kernel
4.12.0
Search docs
The Linux kernel user's and administrator's guide
The Linux kernel user-space API guide
Working with the kernel development community
Development tools for the kernel
How to write kernel documentation
The Linux driver implementer's API guide
⊟ Core API Documentation
⊟ Core utilities
The Linux Kernel API
Generic Associative Array Implementation
⊟ Semantics and Behavior of Atomic and Bitmask Operations
Atomic Type And Operations
Atomic Bitmask
CPU hotplug in the Kernel
Semantics and Behavior of Local Atomic Operations
Concurrency Managed Workqueue (cmwq)
Linux generic IRQ handling
Using flexible arrays in the kernel
Interfaces for kernel debugging
Linux Media Subsystem Documentation
The Linux Input Documentation
Linux GPU Driver Developer's Guide
Security documentation
Linux Sound Subsystem Documentation
Docs » Core API Documentation » Semantics and Behavior of Atomic and Bitmask Operations
View page source

Semantics and Behavior of Atomic and Bitmask Operations

Author: David S. Miller

This document is intended to serve as a guide to Linux port maintainers on how to implement atomic counter, bitops, and spinlock interfaces properly.

Atomic Type And Operations

The atomic_t type should be defined as a signed integer and the atomic_long_t type as a signed long integer. Also, they should be made opaque such that any kind of cast to a normal C integer type will fail. Something like the following should suffice:

```
typedef struct { int counter; } atomic_t;
typedef struct { long counter; } atomic_long_t;
```

Historically, counter has been declared volatile. This is now discouraged. See Documentation/process/volatile-considered-harmful.rst for the complete rationale.

local_t is very similar to atomic_t. If the counter is per CPU and only updated by one CPU, local_t is probably more appropriate. Please see Documentation/core-api/local_ops.rst for the semantics of local_t.

The first operations to implement for atomic_t's are the initializers and plain reads.

```
#define ATOMIC_INIT(i)          { (i) }
#define atomic_set(v, i)        ((v)->counter = (i))
```

The first macro is used in definitions, such as:

```
static atomic_t my_counter = ATOMIC_INIT(1);
```

The initializer is atomic in that the return values of the atomic operations are guaranteed to be correct reflecting the initialized value if the initializer is used before runtime. If the initializer is used at runtime, a proper implicit or explicit read memory barrier is needed before reading the value with atomic_read from another thread.

As with all of the `atomic_` interfaces, replace the leading `atomic_` with `atomic_long_` to operate on atomic_long_t.

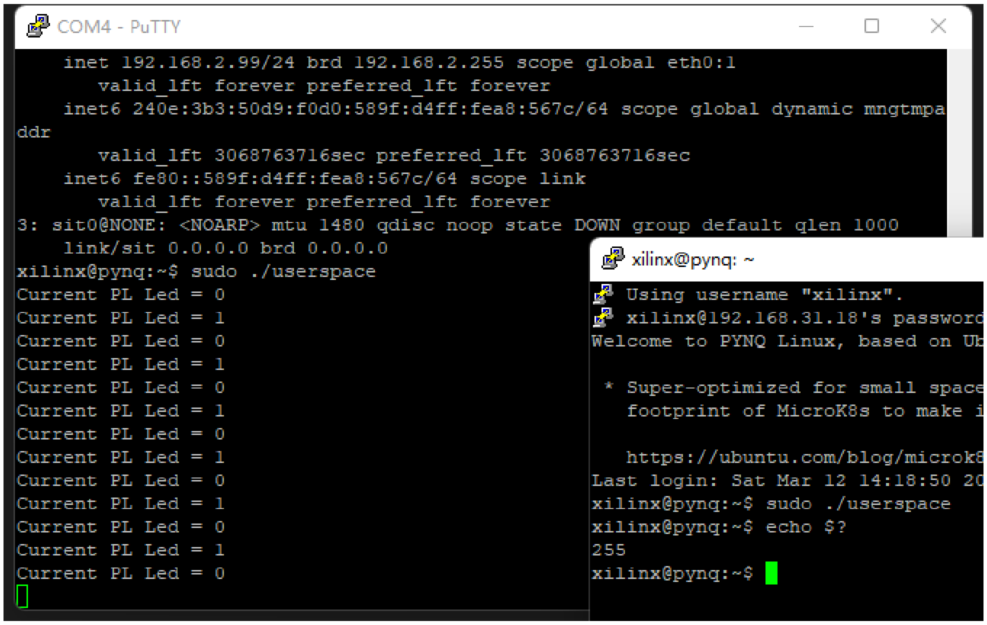
为了使我们的LED驱动受到保护,以原子操作来保护,首先LED只有一个,所以如果我把LED卖给了第一个用户程序,我就不能卖给第二个,除非他释放了,所以我们只有一个LED可以卖,我们从最后一个LED驱动开始改,下面是驱动文件,主要在初始化,打开和释放时候插入了操作,在结构体中新增了一个原子变量.

```
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/cdev.h>
#include <linux/of.h> /* dts操作相关 */
#include <linux/of_address.h> /* dts地址相关 */
#include <linux/of_gpio.h> /* gpio子系统相关 */

#define KERNEL_LED_DEVIE_CNT 1
#define KERNEL_LED_NAME "kernel_led"

#define GPIO_CLK_EN (0x1U << 22)

#define EMIO_PIN 0x00000001

#define EMIO_INPUT 0
#define EMIO_OUTPUT 1

#define EMIO_OUTPUT_DIS 0
#define EMIO_OUTPUT_EN 1

#define EMIO_GPIO_LOW 0
#define EMIO_GPIO_HIGH 1
```

```
struct kernel_led_dev
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int acminor;
    struct device_node *nd; /* 设备节点 */
    int gpio; /* gpio编号 */
    atomic_t lock; /* 原子变量 */
};

static struct kernel_led_dev dev;

/* 设备打开时候会被调用 */
static int led_open(struct inode *inode,struct file *filp){
    /* 不能用atomic_dec再atomic_read,因为多条命令就是破坏原子性,一个功能只有一个原子命令做. */
    if(!atomic_dec_and_test(&dev.lock)){ /* 减1并测试,如果为0,则true,因为默认是1,取走后是0,所以就是有资源,否则去走后可能是负数,说明没资源. */
        atomic_set(&dev.lock,0); /* 刚才没有资源还取走他,所以现在还原为0,为什么不能用atomic_inc?思考一下. */
        return -EBUSY;
    }

    filp->private_data = &dev;

    return 0;
}

/* 设备读取时候会被调用 */
static ssize_t led_read(struct file *filp,char __user *buf,size_t cnt,loff_t *offset){
    int ret;
    char kbuf[1];

    if (cnt != 1){
        return -EFAULT;
    }

    ret = gpio_get_value(dev.gpio);
    if(ret < 0){
        return ret;
    }

    /* 不是高就是低! */
    kbuf[0] = ret;
    ret = copy_to_user(buf,kbuf,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }

    return 0;
}

/* 设备写入时候会被调用 */
static ssize_t led_write(struct file *filp,const char __user *buf,size_t cnt,loff_t *offset){
    int ret;
    char kbuf[1];

    if (cnt != 1){
        return -EFAULT;
    }

    ret = copy_from_user(kbuf,buf,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }

    gpio_set_value(dev.gpio,kbuf[0]?1:0);

    return 0;
}
```

```
/* 设备释放时候会被调用 */
static int led_release(struct inode *inode,struct file *filp){
    atomic_set(&dev.lock,1); /* 释放时候记得归还 */

    return 0;
}

static struct file_operations fops =
{
    .owner = THIS_MODULE,
    .open = led_open,
    .read = led_read,
    .write = led_write,
    .release = led_release,
};

static int __init led_init(void){
    const char *str;
    int ret;

    /* 新增的从dts获取数据的过程 */
    dev.nd = of_find_node_by_path("/led");
    if(dev.nd == NULL){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"status",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"okay")){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"compatible",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"taterli,led")){
        return -EINVAL;
    }

    /* IO当然也可以是一个数组 */
    dev.gpio = of_get_named_gpio(dev.nd,"led-gpio",0);
    if(!gpio_is_valid(dev.gpio)){
        /* IO是独占资源,因此可能申请失败！ */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev.gpio,"taterli-kernel-led");
    if(ret < 0){
        /* 除了返回EINVAL,也可以返回上一层传递的错误． */
        return ret;
    }

    ret = of_property_read_string(dev.nd,"default-state",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(!strcmp(str,"on")){
         /* 设置输出和默认电平 */
        gpio_direction_output(dev.gpio,1);
    }else if(!strcmp(str,"off")){
        gpio_direction_output(dev.gpio,0);
    }else{
        return -EINVAL;
    }

    /* 不需要寄存器映射了,因为有子系统！ */
    /* 申请一个设备号 */
    ret = alloc_chrdev_region(&dev.devid,0,KERNEL_LED_DEVIE_CNT,KERNEL_LED_NAME);
    if(ret){
        goto alloc_fail;
    }
    dev.major = MAJOR(dev.devid);
    dev.minor = MINOR(dev.devid);

    dev.cdev.owner = THIS_MODULE;
    cdev_init(&dev.cdev,&fops);

    ret = cdev_add(&dev.cdev,dev.devid,KERNEL_LED_DEVIE_CNT);
    if(ret){
        goto add_fail;
    }

    dev.class = class_create(THIS_MODULE,KERNEL_LED_NAME);
    if(IS_ERR(dev.class)){
        ret = PTR_ERR(dev.class);
        goto class_fail;
    }

    dev.device = device_create(dev.class,NULL,dev.devid,NULL,KERNEL_LED_NAME);
    if(IS_ERR(dev.device)){
        ret = PTR_ERR(dev.class);
        goto dev_fail;
    }

    atomic_set(&dev.lock,1); /* 初始设置 */

    return 0;

dev_fail:
    class_destroy(dev.class);

class_fail:
    cdev_del(&dev.cdev);

add_fail:
    unregister_chrdev_region(dev.devid,KERNEL_LED_DEVIE_CNT);

alloc_fail:
    /* 这里就清爽很多了,释放IO就行. */
    gpio_free(dev.gpio);
    return ret;
}

static void __exit led_exit(void){
    device_destroy(dev.class,dev.devid);

    class_destroy(dev.class);

    cdev_del(&dev.cdev);

    unregister_chrdev_region(dev.devid,KERNEL_LED_DEVIE_CNT);

    gpio_free(dev.gpio);
}

module_init(led_init);
module_exit(led_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Led GPIO");
MODULE_LICENSE("GPL");
```

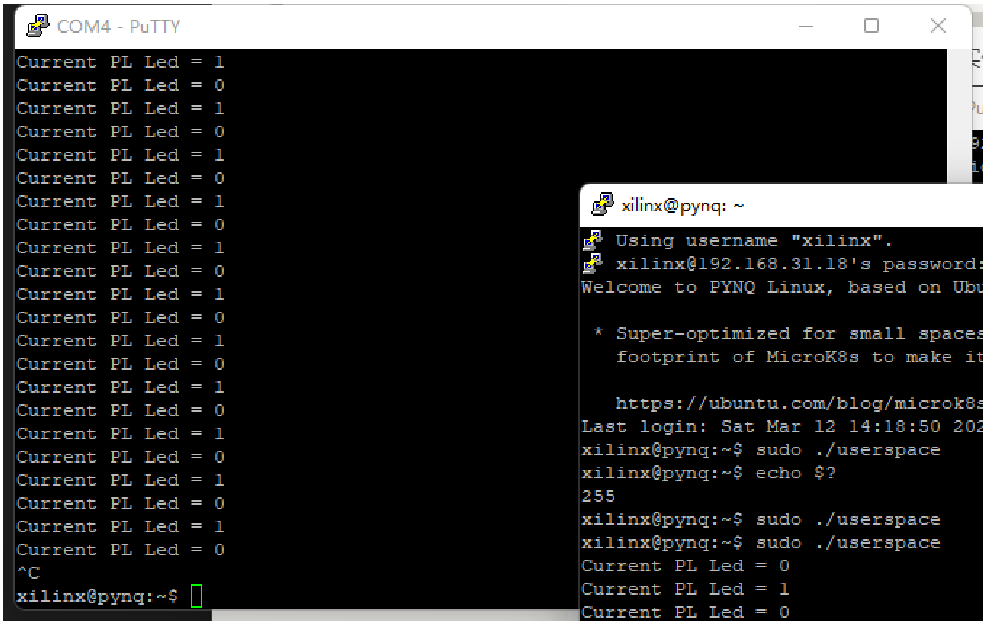
实验中第二个实例打不开了,返回255,其实就是触发到BUSY里了.



```
    inet 192.168.2.99/24 brd 192.168.2.255 scope global eth0:1
       valid_lft forever preferred_lft forever
    inet6 240e:3b3:50d9:f0d0:589f:d4ff:fea8:567c/64 scope global dynamic mngtmpa
ddr
       valid_lft 3068763716sec preferred_lft 3068763716sec
    inet6 fe80::589f:d4ff:fea8:567c/64 scope link
       valid_lft forever preferred_lft forever
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1000
    link/sit 0.0.0.0 brd 0.0.0.0
xilinx@pynq:~$ sudo ./userspace
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
```

```
Using username "xilinx".
xilinx@192.168.31.18's passwor
Welcome to PYNQ Linux, based on Ub

 * Super-optimized for small space
   footprint of MicroK8s to make i

   https://ubuntu.com/blog/microk8
Last login: Sat Mar 12 14:18:50 20
xilinx@pynq:~$ sudo ./userspace
xilinx@pynq:~$ echo $?
255
xilinx@pynq:~$
```

当一边释放后,另一个也可以打开,达到同时只有一个程序用LED的目的.



```
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
^C
xilinx@pynq:~$
```

```
Using username "xilinx".
xilinx@192.168.31.18's password:
Welcome to PYNQ Linux, based on Ubu

 * Super-optimized for small spaces
   footprint of MicroK8s to make it

   https://ubuntu.com/blog/microk8s
Last login: Sat Mar 12 14:18:50 202
xilinx@pynq:~$ sudo ./userspace
xilinx@pynq:~$ echo $?
255
xilinx@pynq:~$ sudo ./userspace
xilinx@pynq:~$ sudo ./userspace
Current PL Led = 0
Current PL Led = 1
Current PL Led = 0
```

自旋锁全部以spin_开头,与原子操作很类似,并且自旋锁有自己的irq上下文保存方法,虽然稍微多占用资源,但是安全很多,而且自旋锁中有很多应用,除了内核上,很多数据库也有自旋锁,比如读写自旋锁,可以并发读但是只能一个人写,顺序锁,每个执行必须严格按照流程排队一个一个执行,例子中用了irq相关的spin函数,兼容性更好,可以避免打断系统中断,但是占用资源也稍微多一些.

```
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
```

```
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/cdev.h>
#include <linux/of.h> /* dts操作相关 */
#include <linux/of_address.h> /* dts地址相关 */
#include <linux/of_gpio.h> /* gpio子系统相关 */

#define KERNEL_LED_DEVIE_CNT 1
#define KERNEL_LED_NAME "kernel_led"

#define GPIO_CLK_EN (0x1U << 22)

#define EMIO_PIN 0x00000001

#define EMIO_INPUT 0
#define EMIO_OUTPUT 1

#define EMIO_OUTPUT_DIS 0
#define EMIO_OUTPUT_EN 1

#define EMIO_GPIO_LOW 0
#define EMIO_GPIO_HIGH 1

struct kernel_led_dev
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int minor;
    struct device_node *nd; /* 设备节点 */
    int gpio; /* gpio编号 */
    spinlock_t lock; /* 自旋锁 */
    int used; /* 有人在用吗 */
};

static struct kernel_led_dev dev;

/* 设备打开时候会被调用 */
static int led_open(struct inode *inode,struct file *filp){
    unsigned long flags;

    spin_lock_irqsave(&dev.lock,flags);
    if(dev.used){
        spin_lock_irqrestore(&dev.lock,flags);
        return -EBUSY;
    }

    dev.used = 1;

    filp->private_data = &dev;
    spin_lock_irqrestore(&dev.lock,flags);

    return 0;
}

/* 设备读取时候会被调用 */
static ssize_t led_read(struct file *filp,char __user *buf,size_t cnt,loff_t *offset){
    int ret;
    char kbuf[1];

    if (cnt != 1){
        return -EFAULT;
    }

    ret = gpio_get_value(dev.gpio);
    if(ret < 0){
        return ret;
    }

    /* 不是高就是低！ */
    kbuf[0] = ret;
    ret = copy_to_user(buf,kbuf,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }

    return 0;
}

/* 设备写入时候会被调用 */
static ssize_t led_write(struct file *filp,const char __user *buf,size_t cnt,loff_t *offset){
    int ret;
    char kbuf[1];

    if (cnt != 1){
        return -EFAULT;
    }

    ret = copy_from_user(kbuf,buf,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }

    gpio_set_value(dev.gpio,kbuf[0]?1:0);

    return 0;
}

/* 设备释放时候会被调用 */
static int led_release(struct inode *inode,struct file *filp){
    unsigned long flags;

    spin_lock_irqsave(&dev.lock,flags);

    dev.used = 0;

    spin_lock_irqrestore(&dev.lock,flags);

    return 0;
}

static struct file_operations fops =
{
    .owner = THIS_MODULE,
    .open = led_open,
    .read = led_read,
    .write = led_write,
    .release = led_release,
};

static int __init led_init(void){
    const char *str;
    int ret;

    /* 新增的从dts获取数据的过程 */
    dev.nd = of_find_node_by_path("/led");
    if(dev.nd == NULL){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"status",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"okay")){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"compatible",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"taterli,led")){
        return -EINVAL;
    }

    /* IO当然也可以是一个数组 */
    dev.gpio = of_get_named_gpio(dev.nd,"led-gpio",0);
    if(!gpio_is_valid(dev.gpio)){
        /* IO是独占资源,因此可能申请失败! */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev.gpio,"taterli-kernel-led");
    if(ret < 0){
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    ret = of_property_read_string(dev.nd,"default-state",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(!strcmp(str,"on")){
         /* 设置输出和默认电平 */
        gpio_direction_output(dev.gpio,1);
    }else if(!strcmp(str,"off")){
        gpio_direction_output(dev.gpio,0);
    }else{
        return -EINVAL;
    }

    /* 不需要寄存器映射了,因为有子系统! */

    /* 申请一个设备号 */
    ret = alloc_chrdev_region(&dev.devid,0,KERNEL_LED_DEVIE_CNT,KERNEL_LED_NAME);
    if(ret){
        goto alloc_fail;
    }
    dev.major = MAJOR(dev.devid);
    dev.minor = MINOR(dev.devid);

    dev.cdev.owner = THIS_MODULE;
    cdev_init(&dev.cdev,&fops);

    ret = cdev_add(&dev.cdev,dev.devid,KERNEL_LED_DEVIE_CNT);
    if(ret){
        goto add_fail;
    }

    dev.class = class_create(THIS_MODULE,KERNEL_LED_NAME);
    if(IS_ERR(dev.class)){
        ret = PTR_ERR(dev.class);
        goto class_fail;
    }

    dev.device = device_create(dev.class,NULL,dev.devid,NULL,KERNEL_LED_NAME);
    if(IS_ERR(dev.device)){
        ret = PTR_ERR(dev.class);
        goto dev_fail;
    }

    dev.used = 0;
    spin_lock_init(&dev.lock); /* 初始设置 */

    return 0;

dev_fail:
    class_destroy(dev.class);

class_fail:
    cdev_del(&dev.cdev);
```

```
add_fail:
    unregister_chrdev_region(dev.devid,KERNEL_LED_DEVIE_CNT);

alloc_fail:
    /* 这里就清爽很多了,释放IO就行. */
    gpio_free(dev.gpio);
    return ret;
}

static void __exit led_exit(void){
    device_destroy(dev.class,dev.devid);

    class_destroy(dev.class);

    cdev_del(&dev.cdev);

    unregister_chrdev_region(dev.devid,KERNEL_LED_DEVIE_CNT);

    gpio_free(dev.gpio);
}

module_init(led_init);
module_exit(led_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Led GPIO");
MODULE_LICENSE("GPL");
```

在上面的程序中,我们可以称used是信号量一样的角色,如果used没有存在的货,将不能继续,我们把它换成真的信号量实现,注意信号量更复杂一些,因此他需要额外引入头文件.

```
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/cdev.h>
#include <linux/of.h> /* dts操作相关 */
#include <linux/of_address.h> /* dts地址相关 */
#include <linux/of_gpio.h> /* gpio子系统相关 */
#include <linux/semaphore.h> /* 信号量头文件 */

#define KERNEL_LED_DEVIE_CNT 1
#define KERNEL_LED_NAME "kernel_led"

#define GPIO_CLK_EN (0x1U << 22)

#define EMIO_PIN 0x00000001

#define EMIO_INPUT 0
#define EMIO_OUTPUT 1

#define EMIO_OUTPUT_DIS 0
#define EMIO_OUTPUT_EN 1

#define EMIO_GPIO_LOW 0
#define EMIO_GPIO_HIGH 1

struct kernel_led_dev
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int minor;
    struct device_node *nd; /* 设备节点 */
    int gpio; /* gpio编号 */
    struct semaphore used; /* 信号量 */
};

static struct kernel_led_dev dev;

/* 设备打开时候会被调用 */
static int led_open(struct inode *inode,struct file *filp){
    /* 获取不到会去休眠不会死等,如果用down()会死等,用户空间看起来没区别都在死等. */
    if(down_interruptible(&dev.used)){
        return -ERESTARTSYS;
    }

    filp->private_data = &dev;

    return 0;
}

/* 设备读取时候会被调用 */
static ssize_t led_read(struct file *filp,char __user *buf,size_t cnt,loff_t *offset){
    int ret;
    char kbuf[1];

    if (cnt != 1){
        return -EFAULT;
    }

    ret = gpio_get_value(dev.gpio);
    if(ret < 0){
        return ret;
    }

    /* 不是高就是低! */
    kbuf[0] = ret;
    ret = copy_to_user(buf,kbuf,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }

    return 0;
}

/* 设备写入时候会被调用 */
static ssize_t led_write(struct file *filp,const char __user *buf,size_t cnt,loff_t *offset){
    int ret;
    char kbuf[1];

    if (cnt != 1){
        return -EFAULT;
    }

    ret = copy_from_user(kbuf,buf,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }

    gpio_set_value(dev.gpio,kbuf[0]?1:0);

    return 0;
}

/* 设备释放时候会被调用 */
static int led_release(struct inode *inode,struct file *filp){
    up(&dev.used);

    return 0;
}

static struct file_operations fops =
{
    .owner = THIS_MODULE,
    .open = led_open,
    .read = led_read,
    .write = led_write,
    .release = led_release,
};

static int __init led_init(void){
    const char *str;
    int ret;

    /* 新增的从dts获取数据的过程 */
    dev.nd = of_find_node_by_path("/led");
    if(dev.nd == NULL){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"status",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"okay")){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"compatible",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"taterli,led")){
        return -EINVAL;
    }

    /* IO当然也可以是一个数组 */
    dev.gpio = of_get_named_gpio(dev.nd,"led-gpio",0);
    if(!gpio_is_valid(dev.gpio)){
        /* IO是独占资源,因此可能申请失败！ */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev.gpio,"taterli-kernel-led");
    if(ret < 0){
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    ret = of_property_read_string(dev.nd,"default-state",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(!strcmp(str,"on")){
         /* 设置输出和默认电平 */
        gpio_direction_output(dev.gpio,1);
    }else if(!strcmp(str,"off")){
        gpio_direction_output(dev.gpio,0);
    }else{
        return -EINVAL;
    }

    /* 不需要寄存器映射了,因为有子系统！ */

    /* 申请一个设备号 */
    ret = alloc_chrdev_region(&dev.devid,0,KERNEL_LED_DEVIE_CNT,KERNEL_LED_NAME);
    if(ret){
        goto alloc_fail;
    }
    dev.major = MAJOR(dev.devid);
    dev.minor = MINOR(dev.devid);
    dev.cdev.owner = THIS_MODULE;
    cdev_init(&dev.cdev,&fops);

    ret = cdev_add(&dev.cdev,dev.devid,KERNEL_LED_DEVIE_CNT);
    if(ret){
        goto add_fail;
    }

    dev.class = class_create(THIS_MODULE,KERNEL_LED_NAME);
    if(IS_ERR(dev.class)){
        ret = PTR_ERR(dev.class);
        goto class_fail;
    }

    dev.device = device_create(dev.class,NULL,dev.devid,NULL,KERNEL_LED_NAME);
    if(IS_ERR(dev.device)){
        ret = PTR_ERR(dev.class);
        goto dev_fail;
    }

    sema_init(&dev.used,1); /* 初始设置 */

    return 0;

dev_fail:
    class_destroy(dev.class);

class_fail:
    cdev_del(&dev.cdev);

add_fail:
    unregister_chrdev_region(dev.devid,KERNEL_LED_DEVIE_CNT);

alloc_fail:
    /* 这里就清爽很多了,释放IO就行. */
    gpio_free(dev.gpio);
    return ret;
}

static void __exit led_exit(void){
    device_destroy(dev.class,dev.devid);

    class_destroy(dev.class);

    cdev_del(&dev.cdev);

    unregister_chrdev_region(dev.devid,KERNEL_LED_DEVIE_CNT);

    gpio_free(dev.gpio);
}

module_init(led_init);
module_exit(led_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Led GPIO");
MODULE_LICENSE("GPL");
```

再简单修改一下,也可以测试一下互斥体的实验.

```
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
```

```
#include <linux/cdev.h>
#include <linux/of.h> /* dts操作相关 */
#include <linux/of_address.h> /* dts地址相关 */
#include <linux/of_gpio.h> /* gpio子系统相关 */
#include <linux/mutex.h> /* 信号量头文件 */

#define KERNEL_LED_DEVIE_CNT 1
#define KERNEL_LED_NAME "kernel_led"

#define GPIO_CLK_EN (0x1U << 22)

#define EMIO_PIN 0x00000001

#define EMIO_INPUT 0
#define EMIO_OUTPUT 1

#define EMIO_OUTPUT_DIS 0
#define EMIO_OUTPUT_EN 1

#define EMIO_GPIO_LOW 0
#define EMIO_GPIO_HIGH 1

struct kernel_led_dev
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int minor;
    struct device_node *nd; /* 设备节点 */
    int gpio; /* gpio编号 */
    struct mutex used; /* 信号量 */
};

static struct kernel_led_dev dev;

/* 设备打开时候会被调用 */
static int led_open(struct inode *inode,struct file *filp){
    /* 获取不到会去休眠不会死等,如果用mutex_lock()会死等,用户空间看起来没区别都在死等. */
    if(mutex_lock_interruptible(&dev.used)){
        return -ERESTARTSYS;
    }

    filp->private_data = &dev;

    return 0;
}

/* 设备读取时候会被调用 */
static ssize_t led_read(struct file *filp,char __user *buf,size_t cnt,loff_t *offset){
    int ret;
    char kbuf[1];

    if (cnt != 1){
        return -EFAULT;
    }

    ret = gpio_get_value(dev.gpio);
    if(ret < 0){
        return ret;
    }

    /* 不是高就是低! */
    kbuf[0] = ret;
    ret = copy_to_user(buf,kbuf,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }

    return 0;
}
```

```
/* 设备写入时候会被调用 */
static ssize_t led_write(struct file *filp,const char __user *buf,size_t cnt,loff_t *offset){
    int ret;
    char kbuf[1];

    if (cnt != 1){
        return -EFAULT;
    }

    ret = copy_from_user(kbuf,buf,cnt);
    if(ret){
        /* 复制失败了 */
        return -EFAULT;
    }

    gpio_set_value(dev.gpio,kbuf[0]?1:0);

    return 0;
}

/* 设备释放时候会被调用 */
static int led_release(struct inode *inode,struct file *filp){
    mutex_unlock(&dev.used);

    return 0;
}

static struct file_operations fops =
{
    .owner = THIS_MODULE,
    .open = led_open,
    .read = led_read,
    .write = led_write,
    .release = led_release,
};

static int __init led_init(void){
    const char *str;
    int ret;

    /* 新增的从dts获取数据的过程 */
    dev.nd = of_find_node_by_path("/led");
    if(dev.nd == NULL){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"status",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"okay")){
        return -EINVAL;
    }

    ret = of_property_read_string(dev.nd,"compatible",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(strcmp(str,"taterli,led")){
        return -EINVAL;
    }

    /* IO当然也可以是一个数组 */
    dev.gpio = of_get_named_gpio(dev.nd,"led-gpio",0);
    if(!gpio_is_valid(dev.gpio)){
        /* IO是独占资源,因此可能申请失败! */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev.gpio,"taterli-kernel-led");
    if(ret < 0){
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    ret = of_property_read_string(dev.nd,"default-state",&str);
    if(ret < 0){
        return -EINVAL;
    }

    if(!strcmp(str,"on")){
         /* 设置输出和默认电平 */
        gpio_direction_output(dev.gpio,1);
    }else if(!strcmp(str,"off")){
        gpio_direction_output(dev.gpio,0);
    }else{
        return -EINVAL;
    }

    /* 不需要寄存器映射了,因为有子系统! */

    /* 申请一个设备号 */
    ret = alloc_chrdev_region(&dev.devid,0,KERNEL_LED_DEVIE_CNT,KERNEL_LED_NAME);
    if(ret){
        goto alloc_fail;
    }
    dev.major = MAJOR(dev.devid);
    dev.minor = MINOR(dev.devid);

    dev.cdev.owner = THIS_MODULE;
    cdev_init(&dev.cdev,&fops);

    ret = cdev_add(&dev.cdev,dev.devid,KERNEL_LED_DEVIE_CNT);
    if(ret){
        goto add_fail;
    }

    dev.class = class_create(THIS_MODULE,KERNEL_LED_NAME);
    if(IS_ERR(dev.class)){
        ret = PTR_ERR(dev.class);
        goto class_fail;
    }

    dev.device = device_create(dev.class,NULL,dev.devid,NULL,KERNEL_LED_NAME);
    if(IS_ERR(dev.device)){
        ret = PTR_ERR(dev.class);
        goto dev_fail;
    }

    mutex_init(&dev.used); /* 初始设置 */

    return 0;

dev_fail:
    class_destroy(dev.class);

class_fail:
    cdev_del(&dev.cdev);

add_fail:
    unregister_chrdev_region(dev.devid,KERNEL_LED_DEVIE_CNT);

alloc_fail:
    /* 这里就清爽很多了,释放IO就行. */
    gpio_free(dev.gpio);
    return ret;
}

static void __exit led_exit(void){
    device_destroy(dev.class,dev.devid);

    class_destroy(dev.class);

    cdev_del(&dev.cdev);
    unregister_chrdev_region(dev.devid,KERNEL_LED_DEVIE_CNT);

    gpio_free(dev.gpio);
}

module_init(led_init);
module_exit(led_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Led GPIO");
MODULE_LICENSE("GPL");
```

现在简单地解决了资源的保护问题,毕竟硬件资源是有限的,并且同时操作可能产生太多不可预期,因此涉及此类资源,应该选择合适的保护方法.