Linux 设备驱动之并发控制(二)

——编写者:草根老师(程姚根)

上一节，我们了解到我们可以通过自旋锁进行并发控制。通过我们对自旋锁的认识，我们知道自旋锁有很多限制。当一个进程没有获到自旋锁的时候，它会内核空间自旋，降低系统的效率。很多时候，我们更希望当一个进程去访问设备的时候，如果发现设备正在被别的进程访问，它就进行休眠直到设备空闲为止。

一、信号量

Linux 内核给我们提供了并发控制的另一种机制"信号量",它和自旋锁类似，只有得到信号量的进程，才可以访问共享资源。不同的是，当获取不到信号量时，进程不会进行自旋而是进入休眠等待状态。

好了，下面我们就来看看Linux 内核空间如何使用信号量吧!

```
#include <linux/semaphore.h>

struct semaphore
{
    //用来对count变量起保护作用
    spinlock_t      lock;
    //资源的个数
    unsigned int        count;
    //存放等待资源而进行休眠的进程
    struct list_head    wait_list;
};
```

1.定义信号量

```
struct semaphore sem;
```

2.初始化信号量

```
//初始化sem 成员变量count 值为val
static inline void sema_init(struct semaphore *sem, int val)
//初始化sem 成员变量count 值为1,用于互斥
#define init_MUTEX(sem)     sema_init(sem, 1)
//初始化sem 成员变量count 值为0
#define init_MUTEX_LOCKED(sem)  sema_init(sem, 0)
```

```
-----------------------------------------------------------------
定义和初始化也可以一步完成:
DECLARE_MUTEX(name);//定义信号量name,并且初始化为 1
DECLARE_MUTEX_LOCKED(name);//定义信号量name,并且初始化为0

注意:在Linux内核中，所有的信号量的值几乎都被初始化为1,用于互斥
-----------------------------------------------------------------
```

3.获得信号量

```
/**
 * down - acquire(获取) the semaphore
 * @sem: the semaphore to be acquired
 *
 * Acquires the semaphore.  If no more tasks are allowed to acquire the
 * semaphore, calling this function will put the task to sleep until the
 * semaphore is released.
 *
 * Use of this function is deprecated(不赞成), please use down_interruptible() or
 * down_killable() instead.
 */
void down(struct semaphore *sem)
功能:获取信号量,当信号量不能获取的时候，让进程进入休眠，不能被信号唤醒
```

```
/**
 * down_interruptible - acquire the semaphore unless interrupted
 * @sem: the semaphore to be acquired
 *
 * Attempts to acquire the semaphore.  If no more tasks are allowed to
 * acquire the semaphore, calling this function will put the task to sleep.
 * If the sleep is interrupted by a signal, this function will return -EINTR.
 * If the semaphore is successfully acquired, this function returns 0.
 */
int down_interruptible(struct semaphore *sem)
功能:获取信号量，当信号量不能获取的时候，让进程进入休眠，可以被信号唤醒
返回值:
成功返回0,失败返回-EINTR
```

```
/**
 * down_killable - acquire the semaphore unless killed
 * @sem: the semaphore to be acquired
 *
```

```
 * Attempts to acquire the semaphore.  If no more tasks are allowed to
 * acquire the semaphore, calling this function will put the task to sleep.
 * If the sleep is interrupted by a fatal signal, this function will return
 * -EINTR.  If the semaphore is successfully acquired, this function returns
 * 0.
 */
int down_killable(struct semaphore *sem)
```

功能：
获取信号量，当信号量不能获取的时候，让进程进入休眠，可以被致命信号唤醒(SIGKILL)

返回值：
成功返回0,失败返回-EINTR

4.释放信号量

```
/**
 * up - release the semaphore
 * @sem: the semaphore to release
 *
 * Release the semaphore.  Unlike mutexes, up() may be called from any
 * context and even by tasks which have never called down().
 */
void up(struct semaphore *sem)
```

嗯！明白了信号量，接下来我们来思考一个问题:

如果一个A进程向我们的dev_fifo中写数据，而另一个进程向我们的dev_fifo中读数据,此时会发生什么情况?
是不是极有可能会出现，A进程要写的数据还没写完，B进程就开始读了或B进程还没有读完，A进程又开始写了。

实际上我们更希望达到的效果:A进程写的时候，B进程不要去读或B进程读的时候 ,A进程不要去写。怎么才能达到这种效果呢?

你可能会想到自旋锁和信号量都可以了，是选择"自旋锁"好，还是选择"信号量"好呢?我觉得使用"信号量"进行并发控制比较好,想想为什么?

下面给出，我用信号量实现的并发控制代码:

```
//读设备
static ssize_t dev_fifo_read(struct file *file, char __user *ubuf, size_t size, loff_t *ppos)
{
    int n;
    int ret;
    char *kbuf;

    printk("read *ppos : %lld\n",*ppos);

    if(down_interruptible(&gcd->sem))
        return -ERESTARTSYS;
    printk("get sem !\n");

    if(*ppos == gcd->len || gcd->len == 0)
        return 0;

    //请求大大小 > buffer剩余的字节数 :读取实际记得字节数
    if(size > gcd->len - *ppos)
        n = gcd->len  - *ppos;
    else
        n = size;

    printk("n = %d\n",n);
    //从上一次文件位置指针的位置开始读取数据
    kbuf = gcd->buffer + *ppos;

    //拷贝数据到用户空间
    ret = copy_to_user(ubuf,kbuf, n);
    if(ret != 0)
        return -EFAULT;

    //更新文件位置指针的值
    *ppos += n;

    up(&gcd->sem);

    printk("dev_fifo_read success!\n");

    return n;
}

//写设备
static ssize_t dev_fifo_write(struct file *file, const char __user *ubuf, size_t size, loff_t *ppos)
{
    int n;
    int ret;
    char *kbuf;
```

```
    printk("write *ppos : %lld\n",*ppos);

    if(down_interruptible(&gcd->sem))
        return -ERESTARTSYS;

    //已经到达buffer尾部了
    if(*ppos == sizeof(gcd->buffer))
        return -1;

    //请求大大小 > buffer剩余的字节数(有多少空间就写多少数据)
    if(size > sizeof(gcd->buffer) - *ppos)
        n = sizeof(gcd->buffer) - *ppos;
    else
        n = size;

    //从上一次文件位置指针的位置开始写入数据
    kbuf = gcd->buffer + *ppos;
```

```
    //拷贝数据到内核空间
    ret = copy_from_user(kbuf, ubuf, n);
    if(ret != 0)
        return -EFAULT;

    //更新文件位置指针的值
    *ppos += n;

    //更新dev_fifo.len
    gcd->len += n;

    up(&gcd->sem);

    printk("dev_fifo_write success!\n");
    return n;
}
```

二、互斥锁

虽然使用信号量，我们就可以实现互斥的功能，但是内核还是给我们提供了效率更高的专门用来进行互斥的机制 "互斥锁"。
它的使用方法，和信号量一样。唯一不同的是，信号量常用于P、V操作，当然也可以用于互斥，而"互斥锁"只是用来进行互斥的。

下面我们来看看,Linux内核给我们提供的接口:

1.定义互斥锁

```
struct mutex mutex_lock;
```

2.初始化互斥锁

```
void mutex_init(struct mutex *lock);
```

3.获得互斥锁

```
/***
 * mutex_lock - acquire the mutex
 * @lock: the mutex to be acquired
 *
 * Lock the mutex exclusively for this task. If the mutex is not
 * available right now, it will sleep until it can get it.
 *
 * The mutex must later on be released by the same task that
 * acquired it. Recursive locking is not allowed. The task
 * may not exit without first unlocking the mutex. Also, kernel
 * memory where the mutex resides mutex must not be freed with
 * the mutex still locked. The mutex must first be initialized
 * (or statically defined) before it can be locked. memset()-ing
 * the mutex to 0 is not allowed.
 *
 * ( The CONFIG_DEBUG_MUTEXES .config option turns on debugging
 *   checks that will enforce the restrictions and will also do
 *   deadlock debugging. )
```

```
 *
 * This function is similar to (but not equivalent to) down().
 */
void mutex_lock(struct mutex *lock)
```

4.释放互斥锁

```
/***
 * mutex_unlock - release the mutex
 * @lock: the mutex to be released
 *
 * Unlock a mutex that has been locked by this task previously.
 *
 * This function must not be used in interrupt context. Unlocking
 * of a not locked mutex is not allowed.
 *
 * This function is similar to (but not equivalent to) up().
 */
void mutex_unlock(struct mutex *lock)
```

三、原子操作

在说原子操作之前，我们先来思考:如何让一个设备只允许打开一次呢?

很简单，我可以这样做:

```
1.定义一个全局变量 : int open_flag = 0;
2.在驱动层的xxx_open函数中写上如下代码
int  dev_fifo_open(struct inode *node,struct file *file)
{
    if(!open_flag)
    {
        open_flag ++;
    }else{
        return -EBUSY;
    }

    return 0;
}
```

注意思考一下，上面这段代码有没有问题?

如果A进程在真准备执行"open_flag ++"的时候，操作系统调度B进程运行，此时B进程开始执行，执行完dev_fifo_open函数之后，操作系统调度A进程运行。由于A进程接着执行open_flag ++。

出现了什么情况?本来是只允许一个进程打开设备的时候，现在A,B两个进程都认为打开了dev_fifo设备。

怎么解决呢?其实解决这个问题很简单，我们只需要让判断open_flag 和 open_flag ++ 两步用一步完成,就可以了。

在Linux 内核中，提供了一种原子操作的机制。所谓的原子操作，就是不能被中断的操作。下面我们来看如何使用原子操作吧!

```
1.定义原子变量

typedef struct {
    int counter;
} atomic_t;

atomic_t v;

2.设置原子变量值
```

```
void atomic_set(atomic_t *v,int i);//设置原子变量的值为i
atomic_t v = ATOMIC_INIT(i);//定义原子变量，并且初始化值为i
```

3.获取原子变量的值

```
#define atomic_read(v)       (*(volatile int *)&(v)->counter)
```

4.原子变量加/减

```
void atomic_add(int i,atomic_t *v);//原子变量的值 + i
void atomic_sub(int i,atomic_t *v);//原子变量的值 - i
```

5.原子变量自增/自减

```
void atomic_inc(atomic_t *v);//原子变量的值 + 1
void atomic_sub(atomic_t *v);//原子变量的值 - 1
```

6.操作并测试

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i,atomic_t *v);
```

以上函数操作后测试原子变量的值是否为0,为0返回true，否则返回false

7.操作并返回

```
int atomic_add_return(int i,atomic_t *v);
int atomic_sub_return(int i,atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

以上函数，操作完后返回原子变量的值

了解完原子操作后，知道该如何真确的实现xxx_open函数了吗,想一想在看下面答案?

```
//定义原子变量，并且初始化1
atomic_t open_flag = ATOMIC_INIT(1);

//打开设备
static int dev_fifo_open(struct inode *inode, struct file *file)
{
    printk("dev_fifo_open success!\n");

    //将原子变量减 - 1,并且测试原子变量值是否为0,为0返回真，否则返回假
    if(!atomic_dec_and_test(&open_flag))
    {
        atomic_inc(&open_flag);
        return -EBUSY;
    }

    return 0;
}
```