在驱动程序中，当多个线程同时访问相同的资源时（驱动程序中的全局变量是一种典型的共享资源，可能会引发“竞态”，因此我们必须对共享资源进行并发控制。

Linux内核中解决并发控制的最常用方法是自旋锁与信号量（绝大多数时候作为互斥锁使用）。

自旋锁与信号量“类似而不类”，类似说的是它们功能上的相似性，“不类”指代它们在本质和实现机理上完全不一样，不属于一类。

自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环查看是否该自旋锁的保持者已经释放了锁，“自旋”就是“在原地打转”。而信号量则引起调用者睡眠，它把进程从运行队列上拖出去，除非获得锁。这就是它们的“不类”。

（sleep和循环执行一万次）

但是，无论是信号量，还是自旋锁，在任何时刻，最多只能有一个保持者，即在任何时刻最多只能有一个执行单元获得锁。这就是它们的“类似”。

鉴于自旋锁与信号量的上述特点，一般而言，自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用；信号量适合于保持时间较长的情况，会只能在进程上下文使用。如果被保护的共享资源只在进程上下文访问，则可以以信号量来保护该共享资源，如果对共享

资源的访问时间非常短，自旋锁也是好的选择。但是，如果被保护的共享资源需要在中断上下文访问（包括底半部即中断处理句柄和顶半部即软中断），就必须使用自旋锁。

与信号量相关的API主要有：

定义信号量

     struct semaphore sem;

初始化信号量

     void sema\_init (struct semaphore \*sem, int val);

该函数初始化信号量，并设置信号量sem的值为val

     void init\_MUTEX (struct semaphore \*sem);

该函数用于初始化一个互斥锁，即它把信号量sem的值设置为1，

等同于sema\_init(struct semaphore \*sem, 1)；

     void init\_MUTEX\_LOCKED (struct semaphore \*sem);

该函数也用于初始化一个互斥锁，但它把信号量sem的值设置为0，等同于sema\_init(struct semaphore \*sem, 0)；

获得信号量

     void down(struct semaphore \* sem);

该函数用于获得信号量sem，它会导致睡眠，因此不能在中断上下文使用；

     int down\_interruptible(struct semaphore \* sem);

该函数功能与down类似，不同之处为，down不能被信号打断，但down\_interruptible能被信号打断；

     int down\_trylock(struct semaphore \* sem);

该函数尝试获得信号量sem，如果能够立刻获得，它就获得该信号量并返回0，否则，返回非0值。它不会导致调用者睡眠，可以在中断上下文使用。

释放信号量

void up(struct semaphore \* sem);

该函数释放信号量sem，唤醒等待者。

与自旋锁相关的API主要有：

定义自旋锁 spinlock\_t spin;

初始化自旋锁 spin\_lock\_init(lock)

该宏用于动态初始化自旋锁lock获得自旋锁

spin\_lock(lock)

该宏用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放；

spin\_trylock(lock)

该宏尝试获得自旋锁lock，如果能立即获得锁，它获得锁并返回真，否则立即返回假，实际上不再“在原地打转”；

释放自旋锁

spin\_unlock(lock) 该宏释放自旋锁lock，它与spin\_trylock或spin\_lock配对使用；

除此之外，还有一组自旋锁使用于中断情况下的API。

例子：首先，在globalvar的驱动程序中，我们可以通过信号量来控制对int global\_var的并发访问

#include <linux/module.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <asm/uaccess.h>

#include <asm/semaphore.h>

MODULE\_LICENSE("GPL");

#define MAJOR\_NUM 254

static ssize\_t globalvar\_read(struct file \*, char \*, size\_t, loff\_t\*);

static ssize\_t globalvar\_write(struct file \*, const char \*, size\_t, loff\_t\*);

struct file\_operations globalvar\_fops =

{

read: globalvar\_read,

write: globalvar\_write,

};

static int global\_var = 0;

static struct semaphore sem;

static int \_\_init globalvar\_init(void)

{

int ret;

ret = register\_chrdev(MAJOR\_NUM, "globalvar", &globalvar\_fops);

if (ret)

{

printk("globalvar register failure");

}

else

{

printk("globalvar register success");

init\_MUTEX(&sem);

}

return ret;

}

static void \_\_exit globalvar\_exit(void)

{

int ret;

ret = unregister\_chrdev(MAJOR\_NUM, "globalvar");

if (ret)

{

printk("globalvar unregister failure");

}

else

{

printk("globalvar unregister success");

}

}

static ssize\_t globalvar\_read(struct file \*filp, char \*buf, size\_t len, loff\_t \*off)

{

//获得信号量

if (down\_interruptible(&sem))

{

return -ERESTARTSYS;

}

//将global\_var从内核空间复制到用户空间

if (copy\_to\_user(buf, &global\_var, sizeof(int)))

{

up(&sem);

return -EFAULT;

}

//释放信号量

up(&sem);

return sizeof(int);

}

ssize\_t globalvar\_write(struct file \*filp, const char \*buf, size\_t len, loff\_t \*off)

{

//获得信号量

if (down\_interruptible(&sem))

{

return -ERESTARTSYS;

}

//将用户空间的数据复制到内核空间的global\_var

if (copy\_from\_user(&global\_var, buf, sizeof(int)))

{

up(&sem);

return -EFAULT;

}

//释放信号量

up(&sem);

return sizeof(int);

}

module\_init(globalvar\_init);

module\_exit(globalvar\_exit);

接下来，我们给globalvar的驱动程序增加open()和release()函数，并在其中借助自旋锁来保护对全局变量int globalvar\_count（记录打开设备的进程数）的访问来实现设备只能被一个进程打开（必须确保globalvar\_count最多只能为1）：

#include <linux/module.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <asm/uaccess.h>

#include <asm/semaphore.h>

MODULE\_LICENSE("GPL");

#define MAJOR\_NUM 254

static ssize\_t globalvar\_read(struct file \*, char \*, size\_t, loff\_t\*);

static ssize\_t globalvar\_write(struct file \*, const char \*, size\_t, loff\_t\*);

static int globalvar\_open(struct inode \*inode, struct file \*filp);

static int globalvar\_release(struct inode \*inode, struct file \*filp);

struct file\_operations globalvar\_fops =

{

read: globalvar\_read,

write: globalvar\_write,

open: globalvar\_open,

release:globalvar\_release,

};

static int global\_var = 0;

static int globalvar\_count = 0;

static struct semaphore sem;

static spinlock\_t spin = SPIN\_LOCK\_UNLOCKED;

static int \_\_init globalvar\_init(void)

{

int ret;

ret = register\_chrdev(MAJOR\_NUM, "globalvar", &globalvar\_fops);

if (ret)

{

printk("globalvar register failure");

}

else

{

printk("globalvar register success");

init\_MUTEX(&sem);

}

return ret;

}

static void \_\_exit globalvar\_exit(void)

{

int ret;

ret = unregister\_chrdev(MAJOR\_NUM, "globalvar");

if (ret)

{

printk("globalvar unregister failure");

}

else

{

printk("globalvar unregister success");

}

}

static int globalvar\_open(struct inode \*inode, struct file \*filp)

{

//获得自选锁

spin\_lock(&spin);

//临界资源访问

if (globalvar\_count)

{

spin\_unlock(&spin);

return -EBUSY;

}

globalvar\_count++;

//释放自选锁

spin\_unlock(&spin);

return 0;

}

static int globalvar\_release(struct inode \*inode, struct file \*filp)

{

globalvar\_count--;

return 0;

}

static ssize\_t globalvar\_read(struct file \*filp, char \*buf, size\_t len, loff\_t \*off)

{

if (down\_interruptible(&sem))

{

return -ERESTARTSYS;

}

if (copy\_to\_user(buf, &global\_var, sizeof(int)))

{

up(&sem);

return -EFAULT;

}

up(&sem);

return sizeof(int);

}

static ssize\_t globalvar\_write(struct file \*filp, const char \*buf, size\_t len, loff\_t \*off)

{

if (down\_interruptible(&sem))

{

return -ERESTARTSYS;

}

if (copy\_from\_user(&global\_var, buf, sizeof(int)))

{

up(&sem);

return -EFAULT;

}

up(&sem);

return sizeof(int);

}

module\_init(globalvar\_init);

module\_exit(globalvar\_exit);

为了上述驱动程序的效果，我们启动两个进程分别打开

/dev/globalvar。在两个终端中调用./globalvartest.o测试程序，当一个进程打开/dev/globalvar后，另外一个进程将打开失败，输出“device open failure”