# Tekkaman Ninja

tekkamanninja.blog.chinaunix.net





tekkamanninj

博客访问： 75907
博文数量： 263
博客积分： 15936
博客等级： 上将
技术积分： 13951
用 户 组： 普通用户
注册时间： 2007-03-27 11:22

加关注 短消息
论坛 加好友

个人简介

Fedora-ARM

文章分类



文章存档



## Linux设备驱动程序学习（3）-并发和竞态

2007-10-27 09:33:47

分类： LINUX

### Linux设备驱动程序学习（3）-并发和竞态

今天进入《Linux设备驱动程序（第3版）》第五章并发和竞态的学习。

对并发的管理是操作系统编程中核心的问题之一。 并发产生竞态，竞态导致共享数据的非法访问。因为竞态是一种极端低可能性的事件，因此程序员往往会忽视竞态。但是在计算机世界中，百万分之一的事件可能没几秒就会发生，而其结果是灾难性的。

### 一、并发及其管理

竞态通常是作为对资源的共享访问结果而产生的。
在设计自己的驱动程序时，第一个要记住的规则是：只要可能，就应该避免资源的共享。若没有并发访问，就不会有竞态。这种思想的最明显的应用是避免使用全局变量。
但是，资源的共享是不可避免的 ，如硬件资源本质上就是共享、指针传递等等。
资源共享的硬性规则：
（1）在单个执行线程之外共享硬件或软件资源的任何时候，因为另外一个线程可能产生对该资源的不一致观察，因此必须显示地管理对该资源的访问。--访问管理的常见技术成为“锁定”或者“互斥”：确保一次只有一个执行线程可操作共享资源。
（2）当内核代码创建了一个可能和其他内核部分共享的对象时，该对象必须在还有其他组件引用自己时保持存在（并正确工作）。对象尚不能正确工作时，不能将其对内核可用。

### 二、信号量和互斥体

一个信号量（semaphore：旗语，信号灯）本质上是一个整数值，它和一对函数联合使用，这一对函数通常称为P和V。希望进入临届区的进程将在相关信号量上调用P；如果信号量的值大于零，则该值会减小一，而进程可以继续。相反，如果信号量的值为零（或更小），进程必须等待知道其他人释放该信号。对信号量的解锁通过调用V完成；该函数增加信号量的值，并在必要时唤醒等待的进程。
当信号量用于互斥时（即避免多个进程同是在一个临界区运行），信号量的值应初始化为1。这种信号量在任何给定时刻只能由单个进程或线程拥有。在这种使用模式下，一个信号量有事也称为一个“互斥体（mutex）”,它是互斥（mutual exclusion）的简称。Linux内核中几乎所有的信号量均用于互斥。
使用信号量，内核代码必须包含<asm/semaphore.h> 。

以下是信号量初始化的方法：

```
/*初始化函数*/
void sema_init(struct semaphore *sem, int val);
```

由于信号量通常被用于互斥模式。所以以下是内核提供的一组辅助函数和宏：

```
/*方法一、声明+初始化宏*/
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);

/*方法二、初始化函数*/
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);

/*带有“_LOCKED”的是将信号量初始化为0，即锁定，允许任何线程访问时必须先解锁。没带的为1。*/
```

P函数为：

```
void down(struct semaphore *sem); /*不推荐使用，会建立不可杀进程*/
int down_interruptible(struct semaphore *sem);/*推荐使用，使用down_interruptible需要格外小心，若操作被中断，该函数会返回非零值，而调用这不会拥有该信号量。对down_interruptible的正确使用需要始终检查返回值，并做出相应的响应。*/
int down_trylock(struct semaphore *sem);/*带有“_trylock”的永不休眠，若信号量在调用是不可获得，会返回非零值。*/
```

V函数为：

```
void up(struct semaphore *sem);/*任何拿到信号量的线程都必须通过一次（只有一次）对up的调用而释放该信号量。在出错时，要特别小心；若在拥有一个信号量时发生错误，必须在将错误状态返回前释放信号量。*/
```

## 在scull中使用信号量

其实在之前的实验中已经用到了信号量的代码，在这里提一下应该注意的地方：
在初始化scull_dev的地方：

```
/* Initialize each device. */
    for (i = 0; i < scull_nr_devs; i++) {
        scull_devices[i].quantum = scull_quantum;
        scull_devices[i].qset = scull_qset;
        init_MUTEX(&scull_devices[i].sem);/* 注意顺序：先初始化好互斥信号量 ，再使scull_devices可用。*/
        scull_setup_cdev(&scull_devices[i], i);
    }
```

而且要确保在不拥有信号量的时候不会访问scull_dev结构体。

## 读取者/写入者信号量

只读任务可并行完成它们的工作，而不需要等待其他读取者退出临界区。Linux内核提供了读取者/写入者信号量“rwsem”，使用是必须包括<linux/rwsem.h>。
初始化：

```
void init_rwsem(struct rw_semaphore *sem);
```

只读接口：

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

写入接口：

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);

void downgrade_write(struct rw_semaphore *sem);/*该函数用于把写者降级为读者，这有时是必要的。因为写者是排他性的，因此在写者保持读写信号量期间，任何读者或写者都将无法访问该读写信号量保护的共享资源，对于那些当前条件下不需要写访问的写者，降级为读者将，使得等待访问的读者能够立刻访问，从而增加了并发性，提高了效率。*/
```

一个 rwsem 允许一个写者或无限多个读者来拥有该信号量. 写者有优先权; 当某个写者试图进入临界区, 就不会允许读者进入直到写者完成了它的工作. 如果有大量的写者竞争该信号量, 则这个实现可能导致读者“饿死”, 即可能会长期拒绝读者访问。因此, rwsem 最好用在很少请求写的时候, 并且写者只占用短时间.

## completion

completion是一种轻量级的机制，它允许一个线程告诉另一个线程某个工作已经完成。代码必须包含<linux/completion.h>。使用的代码如下：

```
DECLARE_COMPLETION(my_completion);/* 创建completion（声明+初始化）
*/

////////////////////////////////////////////////////

struct completion my_completion;/* 动态声明completion 结构体*/
static inline void init_completion(&my_completion);/*动态初始化
completion*/

////////////////////////////////////////////////////

void wait_for_completion(struct completion *c);/* 等待completion */
void complete(struct completion *c);/*唤醒一个等待completion的线程*/
void complete_all(struct completion *c);/*唤醒所有等待completion的线
程*/

/*如果未使用completion_all，completion可重复使用；否则必须使用以下函
数重新初始化completion*/
INIT_COMPLETION(struct completion c);/*快速重新初始化completion*/
```

completion的典型应用是模块退出时的内核线程终止。在这种远行中，某些驱动程序的内部工作有一个内核线程在while(1)循环中完成。当内核准备清楚该模块时，exit函数会告诉该线程退出并等待completion。为此内核包含了用于这种线程的一个特殊函数：

```
void complete_and_exit(struct completion *c, long retval);
```

## 三、自旋锁

其实上面介绍的几种信号量和互斥机制，其底层源码对于自身结构体的某些变量的维护也用到了现在我们讲到的自旋锁,但是绝不是自旋锁的再包装。

自旋锁是一个互斥设备，他只能会两个值：“锁定”和“解锁”。它通常实现为某个整数之中的单个位。“测试并设置”的操作必须以原子方式完成。

任何时候，只要内核代码拥有自旋锁，在相关CPU上的抢占就会被禁止。

适用于自旋锁的核心规则：

（1）任何拥有自旋锁的代码都必须使原子的，除服务中断外（某些情况下也不能放弃CPU,如中断服务也要获得自旋锁。为了避免这种锁陷阱，需要在拥有自旋锁时禁止中断），不能放弃CPU（如休眠，休眠可发生在许多无法预期的地方）。否则CPU将有可能永远自旋下去（死机）。

（2）拥有自旋锁的时间越短越好。

自旋锁原语所需包含的文件是<linux/spinlock.h>，以下是自旋锁的内核API:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;/* 编译时初始化spinlock*/
void spin_lock_init(spinlock_t *lock);/* 运行时初始化spinlock*/

/* 所有spinlock等待本质上是不可中断的，一旦调用spin_lock，在获得锁之前一直处于自旋状态
*/
void spin_lock(spinlock_t *lock);/* 获得spinlock*/
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);/* 获得spinlock，禁止本
地cpu中断，保存中断标志于flags*/
void spin_lock_irq(spinlock_t *lock);/* 获得spinlock,禁止本地cpu中断*/
void spin_lock_bh(spinlock_t *lock)/* 获得spinlock,禁止软件中断，保持硬件中断打开*/

/* 以下是对应的锁释放函数*/
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

```
/* 以下非阻塞自旋锁函数，成功获得，返回非零值；否则返回零*/
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);

/*新内核的<linux/spinlock.h>包含了更多函数*/
```

读取者/写入者自旋锁:

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED;/* 编译时初始化*/

rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* 运行时初始化*/

void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);

/* 新内核已经有了read_trylock*/

void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);

void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);

/*新内核的<linux/spinlock.h>包含了更多函数*/
```

## 锁陷阱

锁定模式必须在一开始就安排好，否则其后的改进将会非常困难。

不明确规则：如果某个获得锁的函数要调用其他同样试图获取这个锁的函数，代码就会锁死。（不允许锁的拥有者第二次获得同个锁。）为了锁的正确工作，不得不编写一些函数，这些函数假定调用这已经获得了相关的锁。

锁的顺序规则：再必须获取多个锁时，应始终以相同顺序获取。

若必须获得一个局部锁和一个属于内核更中心位置的锁，应先获得局部锁。

若我们拥有信号量和自旋锁的组合，必须先获得信号量。

不得再拥有自旋锁时调用down。（可导致休眠）

尽量避免需要多个锁的情况。

细颗粒度和粗颗粒度的对比：应该在最初使用粗颗粒度的锁，除非有真正的原因相信竞争会导致问题。

## 四、锁之外的办法

### （1）免锁算法

经常用于免锁的生产者/消费者任务的数据结构之一是循环缓冲区。它在设备驱动程序中相当普遍，如以前移植的网卡驱动程序。内核里有一个通用的循环缓冲区的实现在 <linux/kfifo.h> 。

### （2）原子变量

完整的锁机制对一个简单的整数来讲显得浪费。内核提供了一种原子的整数类型，称为atomic_t，定义在<asm/atomic.h>。原子变量操作是非常快的， 因为它们在任何可能时编译成一条单个机器指令。
以下是其接口函数：

```
void atomic_set(atomic_t *v, int i); /*设置原子变量 v 为整数值 i.*/
atomic_t v = ATOMIC_INIT(0);  /*编译时使用宏定义 ATOMIC_INIT 初始化原子值.*/

int atomic_read(atomic_t *v); /*返回 v 的当前值.*/

void atomic_add(int i, atomic_t *v);/*由 v 指向的原子变量加 i. 返回值是 void*/
void atomic_sub(int i, atomic_t *v); /*从 *v 减去 i.*/

void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v); /*递增或递减一个原子变量.*/

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);
/*进行一个特定的操作并且测试结果; 如果, 在操作后, 原子值是 0, 那么返回值是真; 否则, 它是假. 注意没有 atomic_add_and_test.*/

int atomic_add_negative(int i, atomic_t *v);
/*加整数变量 i 到 v. 如果结果是负值返回值是真, 否则为假.*/

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);
/*像 atomic_add 和其类似函数, 除了它们返回原子变量的新值给调用者.*/
```

atomic_t 数据项必须通过这些函数存取。 如果你传递一个原子项给一个期望一个整数参数的函数， 你会得到一个编译错误。需要多个 atomic_t 变量的操作仍然需要某种其他种类的加锁。

### （3）位操作

内核提供了一套函数来原子地修改或测试单个位。原子位操作非常快, 因为它们使用单个机器指令来进行操作, 而在任何时候低层平台做的时候不用禁止中断. 函数是体系依赖的并且在 <asm/bitops.h> 中声明. 以下函数中的数据是体系依赖的. nr 参数(描述要操作哪个位)在ARM体系中定义为unsigned int：

```
void set_bit(nr, void *addr); /*设置第 nr 位在 addr 指向的数据项中。*/

void clear_bit(nr, void *addr); /*清除指定位在 addr 处的无符号长型数据.*/

void change_bit(nr, void *addr);/*翻转nr位.*/

test_bit(nr, void *addr); /*这个函数是唯一一个不需要是原子的位操作; 它简单地返回这个位的当前值.*/
/*以下原子操作如同前面列出的, 除了它们还返回这个位以前的值.*/
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

以下是一个使用范例：

```
/* try to set lock */
while (test_and_set_bit(nr, addr) != 0)
    wait_for_a_while();

/* do your work */

/* release lock, and check. */
if (test_and_clear_bit(nr, addr) == 0)
    something_went_wrong(); /* already released: error */
```

（4）seqlock

2.6内核包含了一对新机制打算来提供快速地， 无锁地存取一个共享资源。 seqlock要保护的资源小，简单， 并且常常被存取， 并且很少写存取但是必须要快。seqlock 通常不能用在保护包含指针的数据结构。seqlock 定义在 <linux/seqlock.h> 。

```
/*两种初始化方法*/
seqlock_t lock1 = SEQLOCK_UNLOCKED;

seqlock_t lock2;
seqlock_init(&lock2);
```

这个类型的锁常常用在保护某种简单计算,读存取通过在进入临界区入口获取一个(无符号的)整数序列来工作. 在退出时, 那个序列值与当前值比较; 如果不匹配, 读存取必须重试.读者代码形式:

```
unsigned int seq;
do {
    seq = read_seqbegin(&the_lock);
    /* Do what you need to do */
} while read_seqretry(&the_lock, seq);
```

如果你的 seqlock 可能从一个中断处理里存取, 你应当使用 IRQ 安全的版本来代替:

```
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned long flags);
```

写者必须获取一个排他锁来进入由一个 seqlock 保护的临界区，写锁由一个自旋锁实现， 调用:

```
void write_seqlock(seqlock_t *lock);
void write_sequnlock(seqlock_t *lock);
```

因为自旋锁用来控制写存取, 所有通常的变体都可用:

```
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);

void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

还有一个 write_tryseqlock 在它能够获得锁时返回非零.

（5）读取-复制-更新

读取-拷贝-更新(RCU) 是一个高级的互斥方法, 在合适的情况下能够有高效率. 它在驱动中的使用很少。

## 五、开发板实验

在我的SBC2440V4开发板上作completion的实验，因为别的实验都要在并发状态下才可以实验，所以本章的我只做了completion的实验。我将《Linux设备驱动程序（第3版）》提供的源码做了修改，将原来的2.4内核的模块接口改成了2.6的接口，并编写了测试程序。实验源码如下：

模块程序链接：complete模块
模块测试程序链接：测试程序

```
[Tekkaman2440@SBC2440V4]#cd /lib/modules/
[Tekkaman2440@SBC2440V4]#insmod complete.ko
[Tekkaman2440@SBC2440V4]#echo 8 > /proc/sys/kernel/printk
[Tekkaman2440@SBC2440V4]#cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  4 ttyS
```

```
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 14 sound
 81 video4linux
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
204 s3c2410_serial
252 complete
253 usb_endpoint
254 rtc

Block devices:
  1 ramdisk
256 rfd
  7 loop
 31 mtdblock
 93 nftl
 96 inftl
179 mmc
[Tekkaman2440@SBC2440V4]#mknod -m 666 /dev/complete c 252 0
[Tekkaman2440@SBC2440V4]#cd /tmp/
[Tekkaman2440@SBC2440V4]#./completion_testr&
[Tekkaman2440@SBC2440V4]#process 814 (completion_test) going to sleep
[Tekkaman2440@SBC2440V4]#./completion_testr&
[Tekkaman2440@SBC2440V4]#process 815 (completion_test) going to sleep
[Tekkaman2440@SBC2440V4]#ps
  PID Uid VSZ Stat Command
    1 root 1744 S init
    2 root SW< [kthreadd]
    3 root SWN [ksoftirqd/0]
    4 root SW< [watchdog/0]
    5 root SW< [events/0]
    6 root SW< [khelper]
   59 root SW< [kblockd/0]
   60 root SW< [ksuspend_usbd]
   63 root SW< [khubd]
   65 root SW< [kseriod]
   77 root SW [pdflush]
   78 root SW [pdflush]
   79 root SW< [kswapd0]
   80 root SW< [aio/0]
  707 root SW< [mtdblockd]
  708 root SW< [nftld]
  709 root SW< [inftld]
  710 root SW< [rfdd]
  742 root SW< [kpsmoused]
  751 root SW< [kmmcd]
  769 root SW< [rpciod/0]
  778 root 1752 S -sh
  779 root 1744 S init
  781 root 1744 S init
```

```
  782 root 1744 S init
  783 root 1744 S init
  814 root 1336 D ./completion_testr
  815 root 1336 D ./completion_testr
  816 root 1744 R ps
[Tekkaman2440@SBC2440V4]#./completion_testw
process 817 (completion_test) awakening the readers...
awoken 814 (completion_test)
write code=0
[Tekkaman2440@SBC2440V4]#read code=0
[Tekkaman2440@SBC2440V4]#ps
  PID Uid VSZ Stat Command
    1 root 1744 S init
    2 root SW< [kthreadd]
    3 root SWN [ksoftirqd/0]
    4 root SW< [watchdog/0]
    5 root SW< [events/0]
    6 root SW< [khelper]
   59 root SW< [kblockd/0]
   60 root SW< [ksuspend_usbd]
   63 root SW< [khubd]
   65 root SW< [kseriod]
   77 root SW [pdflush]
   78 root SW [pdflush]
   79 root SW< [kswapd0]
   80 root SW< [aio/0]
  707 root SW< [mtdblockd]
  708 root SW< [nftld]
  709 root SW< [inftld]
  710 root SW< [rfdd]
  742 root SW< [kpsmoused]
  751 root SW< [kmmcd]
  769 root SW< [rpciod/0]
  778 root 1752 S -sh
  779 root 1744 S init
  781 root 1744 S init
  782 root 1744 S init
  783 root 1744 S init
  815 root 1336 D ./completion_testr
  818 root 1744 R ps
[1] - Done ./completion_testr
[Tekkaman2440@SBC2440V4]#./completion_testw
process 819 (completion_test) awakening the readers...
awoken 815 (completion_test)
write code=0
[Tekkaman2440@SBC2440V4]#read code=0
[Tekkaman2440@SBC2440V4]#ps
  PID Uid VSZ Stat Command
    1 root 1744 S init
    2 root SW< [kthreadd]
    3 root SWN [ksoftirqd/0]
    4 root SW< [watchdog/0]
    5 root SW< [events/0]
    6 root SW< [khelper]
   59 root SW< [kblockd/0]
   60 root SW< [ksuspend_usbd]
   63 root SW< [khubd]
   65 root SW< [kseriod]
   77 root SW [pdflush]
   78 root SW [pdflush]
```

```
  79 root SW< [kswapd0]
  80 root SW< [aio/0]
 707 root SW< [mtdblockd]
 708 root SW< [nftld]
 709 root SW< [inftld]
 710 root SW< [rfdd]
 742 root SW< [kpsmoused]
 751 root SW< [kmmcd]
 769 root SW< [rpciod/0]
 778 root 1752 S -sh
 779 root 1744 S init
 781 root 1744 S init
 782 root 1744 S init
 783 root 1744 S init
 820 root 1744 R ps
[2] + Done ./completion_testr
[Tekkaman2440@SBC2440V4]#ps
  PID Uid VSZ Stat Command
    1 root 1744 S init
    2 root SW< [kthreadd]
    3 root SWN [ksoftirqd/0]
    4 root SW< [watchdog/0]
    5 root SW< [events/0]
    6 root SW< [khelper]
   59 root SW< [kblockd/0]
   60 root SW< [ksuspend_usbd]
   63 root SW< [khubd]
   65 root SW< [kseriod]
   77 root SW [pdflush]
   78 root SW [pdflush]
   79 root SW< [kswapd0]
   80 root SW< [aio/0]
  707 root SW< [mtdblockd]
  708 root SW< [nftld]
  709 root SW< [inftld]
  710 root SW< [rfdd]
  742 root SW< [kpsmoused]
  751 root SW< [kmmcd]
  769 root SW< [rpciod/0]
  778 root 1752 S -sh
  779 root 1744 S init
  781 root 1744 S init
  782 root 1744 S init
  783 root 1744 S init
  821 root 1744 R ps
[Tekkaman2440@SBC2440V4]#./completion_testw
process 822 (completion_test) awakening the readers...
write code=0
[Tekkaman2440@SBC2440V4]#./completion_testr
process 823 (completion_test) going to sleep
awoken 823 (completion_test)
read code=0
```

实验表明：如果先读数据，读的程序会被阻塞（因为驱动在wait_for_completion，等待写的完成）。如果先写，读程序会比较顺利的执行下去（虽然也会休眠，但马上会被唤醒！）。其原因可以从completion的源码中找答案。completion其实就是自旋锁的再包装，具体细节参见completion的源码。

上一篇：Linux设备驱动程序学习（2）-调试技术
下一篇：移植U一Boot.1.2.0到友善之臂SBC2440V4（S3C2440AL）

0

相关热门文章

oracle学习笔记
linux 常见服务端口
移植 ushare 到开发板
[雯]学习策略的方法[p]宜宾...
【ROOTFS搭建】busybox的httpd...
系统提供的库函数存在内存泄漏...
怎么破解别人邮箱密码...
xmanager 2.0 for linux配置
linux虚拟机 求教
linux下安装软件心得
什么是shell
初学UNIX环境高级编程的，关于...
Linux设备驱动程序学习...
linux socket的bug??
chinaunix博客什么时候可以设...

给主人留下些什么吧！~~

评论热议

请登录后评论。
登录 注册