华 清 远 见

FARSIGHT

嵌 入 式 培 训 专 家

Linux驱动程序的并发控制

李安强

版权

- 华清远见嵌入式培训中心版权所有；
- 未经华清远见明确许可，不能为任何目的以任何形式复制或传播此文档的任何部分；
- 本文档包含的信息如有更改，恕不另行通知；
- 保留所有权利。

并发与竞态

- 并发：多个执行单元同时被执行。
- 竞态：并发的执行单元对共享资源（硬件资源和软件上的全局变量等）的访问导致的竞争状态

- 临界资源：
  任何时候只允许一个进程使用的资源为临界资源

  临界区：
  访问临界资源的代码段为临界区

竞态产生的原因

- 对称多处理器（SMP）的多个CPU

- SMP是一种紧耦合,共享存储的系统模型，它的特点是多个CPU使用共同的系统总线，因此可访问共同的外设和存储器

- 单CPU内进程与抢占它的进程

- Linux2.6内核支持抢占调度，一个进程在内核执行的时候被另一高优先级的进程打断，进程与抢占它的进程访问共享资源的情况类似于SMP

- 中断（硬中断、软中断、Tasklet、底半部）与进程之间

- 中断可以打断正在执行的进程，如果中断处理程序访问进程正在访问的资源，则竞态也会发生
- 中断也有可能被新的更高级优先级中断中断，因此，多个中断之间本身也可能引起竞态

并发与竞态

例：

```
if (copy_from_user(&(dev->data[pos]), buf, count))
   ret = -EFAULT;
   goto out;
```

假设有 2 个进程试图同时向一个设备的相同位置写入数据，就会造成数据混乱。

一个设备被多个进程打开

并发与竞态

处理并发的常用技术是加锁或者互斥，即确保在任何时间只有一个执行单元可以操作共享资源。在Linux内核中主要通过semaphore机制和spin_lock机制实现。

中断屏蔽

- 可以保证正在执行的内核执行路径不被中断处理程序抢占,由于Linux内核的进程调度都依赖中断来实现，内核抢占进程之间的竞态就不存在了

- 通过禁止中断的手段，排除单处理器上的并发，会导致中断延迟

- 使用方法:
  - local_irq_disable() //屏蔽中断
  - critical section //临界区
  - local_irq_enable() //开中断

- 说明：
  - local_irq_disable()和local_irq_enable()都只能禁止和使能本CPU内的中断，并不能解决SMP多CPU引发的竞争。

- 缺点:
- 由于Linux系统的异步I/O,进程调度等很多重要操作都依赖于中断，在屏蔽中断期间所有的中断都无法处理，因此长时间屏蔽中断是很危险的，有可能造成数据丢失甚至系统崩溃。

使用场合

- 1)中断与正常进程共享数据
  2)多个中断共享数据
  3)临界区一般很短

原子操作

- 原子操作保证所有指令以原子方式执行(执行过程不能被中断)

整形原子操作

- 针对整数的原子操作只能对atomic_t类型的数据进行处理，在这里之所以引入了一个特殊的数据类型，而没有直接使用C语言的int型，主要是出于两个原因：
- 1）让原子函数只接受atomic_t类型的操作数，可以确保原子操作只与这种特殊类型数据一起使用，同时，这也确保了该类型的数据不会被传递给其它任何非原子函数；
- 2）使用atomic_t类型确保编译器不对相应的值进行访问优化——这点使得原子操作最终接收到正确的内存地址，而不是一个别名，最后就是在不同体系结构上实现原子操作的时候，使用atomic_t可以屏蔽其间的差异。

- 1)设置原子变量的值
- void atomic_set(atomic_t *v, int i);//设置原子变量的值为i
- atomic_t v = ATOMIC_INIT(0);//定义原子变量v并初始化为0

2)获取原子变量的值

atomic_read(atomic_t *v);//返回原子变量的值

- 3)原子变量加/减
- void atomic_add(int i,atomic_t *v); //原子变量增加i
- void atomic_sub(int i,atomic_t *v); //原子变量减少i
- 4)原子变量自增/自减
- void atomic_inc(atomic_t *v); //原子变量加1
- void atomic_dec(atomic_t *v); //原子变量减1

- 5)操作并测试
- int atomic_inc_and_test(atomic_t *v);//这些操作对原子变量执行自增,自减,减操作后测试是否为0,是返回true,否则返回false
- int atomic_dec_and_test(atomic_t *v);
- int atomic_sub_and_test(int i, atomic_t *v);

- 6)操作并返回
- int atomic_add_return(int i,atomic_t *v); //这些操作对原子变量进行对应操作，并返回新的值。
- int atomic_sub_return(int i, atomic_t *v);
- int atomic_inc_return(atomic *v);
- int atomic_dec_return(atomic_t *v);

位原子操作

- 针对位这一级数据进行操作的函数，是对普通的内存地址进行操作的。它的参数是一个指针和一个位号。

- 1)设置位
- void set_bit(nr, void *addr); //设置addr地址的第nr位，所谓设置位即将位写为1
- 2)清除位
- void clear_bit(nr,void *addr); //清除addr地址的第nr位，所谓清除位即将位写为0
- 3)改变位
- void change_bit(nr,void *addr); //对addr地址的第nr位反置

- 4)测试位
- void test_bit(nr, void *addr); //返回addr地址的第nr位
- 5)测试并操作位
- int test_and_set_bit(nr, void *addr);
- int test_and_clear_bit(nr, void *addr);
- int test_and_change_bit(nr, void *addr);

原子变量的使用实例

- 原子变量的使用实例：最多只能被一个进程打开

```
static atomic_t xxx_available = ATOMIC_INIT(1); /*定义原子变量*/
    static int xxx_open(struct inode *inode, struct file *filp)
    {    ...
         if (!atomic_dec_and_test(&xxx_available))
         {
              atomic_inc(&xxx_available);
              return  - EBUSY; /*已经打开*/
         }...
         return 0; /* 成功 */
     }
     static int xxx_release(struct inode *inode, struct file *filp)
     {
          atomic_inc(&xxx_available); /* 释放设备 */
          return 0;
     }
```

使用场合

- 1)共享简单的数据类型：整型，比特性
  2)适合高效率的场合

自旋锁

- 自旋锁（spin lock）是一种对临界资源进行互斥手访问的典型手段

- CPU上将要执行的代码将会执行一个测试并设置某个内存变量的原子操作,若测试结果表明锁已经空闲,则程序获得这个自旋 锁继续运行;若仍被占用,则程序将在一个小的循环内重复测试这个“测试并设置”的操作.这就是自旋。

- 自旋锁就是针对SMP或单个CPU但内核可抢占的情况，对于单CPU和内核不可抢占的系统，自旋锁退化为空操作。
- 还有就是自旋锁解决了临界区不受别的CPU和本CPU内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候还可能受到中断和底半部的影响。

- Linux系统中与自旋锁相关的操作主要有如下4种。
  - 1．定义自旋锁
    - spinlock_t spin;
  - 2．初始化自旋锁
    - spin_lock_init(lock)
  - 3．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”；
  - 4．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。

自旋锁使用举例

- 使用自旋锁使设备只能被一个进程打开

```
int xxx_count = 0;/*定义文件打开次数计数*/
 static int xxx_open(struct inode *inode, struct file *filp)
 { ...
     spinlock(&xxx_lock);
    if (xxx_count)/*已经打开*/
    {   spin_unlock(&xxx_lock);
        return  - EBUSY;
     }
     xxx_count++;/*增加使用计数*/
     spin_unlock(&xxx_lock);
     ...
    return 0; /* 成功 */
 }
 static int xxx_release(struct inode *inode, struct file *filp)
 { ...
    spinlock(&xxx_lock);
    xxx_count--; /*减少使用计数*/
    spin_unlock(&xxx_lock);
    return 0;
}
```

使用场合

- 1)用于多处理器间共享数据
  2)在可抢占的内核线程里共享数据
  3)自旋锁适合于保持时间非常短的情况，

信号量

- 信号量（semaphore）是用于保护临界区的一种常用方法，它的使用方式和自旋锁类似。与自旋锁相同，只有得到信号量的进程才能执行临界区代码。但是，与自旋锁不同的是，当获取不到信号量时，进程不会原地打转而是进入休眠等待状态

信号量相关操作

- 定义信号量
  - struct semaphore sem;
- 初始化信号量
  - void sema_init (struct semaphore *sem, int val);
  - void init_MUTEX(struct semaphore *sem);//初始化为0
- 获得信号量
  - void down(struct semaphore * sem);
  - int down_interruptible(struct semaphore * sem);
  - int down_trylock(struct semaphore * sem);
- 释放信号量
  - void up(struct semaphore * sem);

信号量使用实例

- 使用信号量实现设备只能被一个进程打开

```
 static DECLARE_MUTEX(xxx_lock);//定义互斥锁
static int xxx_open(struct inode *inode, struct file *filp)
 {
          ...
    if (down_trylock(&xxx_lock))  //获得打开锁
       return  - EBUSY;  //设备忙
            ...
    return 0; /* 成功 */
 }
 static int xxx_release(struct inode *inode, struct file *filp)
{
      up(&xxx_lock);  //释放打开锁
     return 0;
}
```

使用场合

- 1)适合于共享区保持时间较长的情况
  2)只能用于进程上下文

自旋锁vs信号量

- 自旋锁和信号量选用的3项原则
  - 1、判断进程切换时间Tsw，和等待获取自旋锁（由临界区执行时间决定）Tcs。如果若Tcs比较小，应使用自旋锁，若Tcs很大，应使用信号量。
  - 2、信号量所保护的临界区可包含可能引起阻塞的代码，而自旋锁则绝对要避免用来保护包含这样代码的临界区。因为阻塞意味着要进行进程的切换，如果进程被切换出去后，另一个进程企图获取本自旋锁，死锁就会发生
  - 3、信号量存在于进程上下文，因此，如果被保护的共享资源需要在中断或软中断情况下使用，则在信号量和自旋锁之间只能选择自旋锁。当然，如果一定要使用信号量，则只能通过down_trylock()方式进行，不能获取就立即返回以避免阻塞。

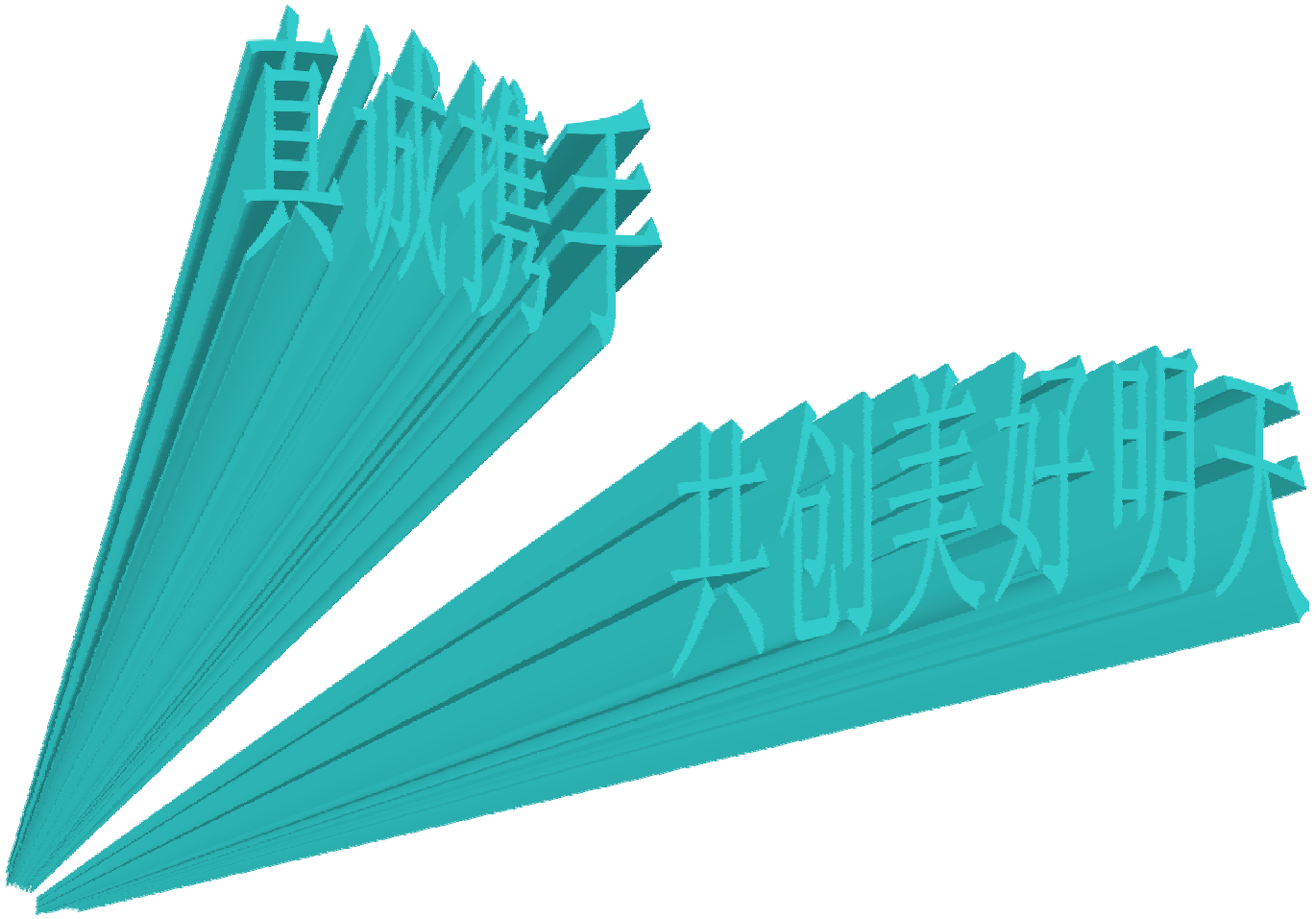
```
}
}
}
```

实例分析

华清远见培训相关课程

- 嵌入式Linux工程师就业班
- 3G Android系统开发就业班
- 嵌入式Linux应用开发
- 嵌入式Linux驱动开发班

}


真诚携手
共创美好明天