华山大师兄

信号量、互斥体和自旋锁

一、信号量

信号量又称为信号灯，它是用来协调不同进程间的数据对象的，而最主要的应用是共享内存方式的进程间通信。本质上，信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。一般说来，为了获得共享资源，进程需要执行下列操作：

（1） 测试控制该资源的信号量。

（2） 若此信号量的值为正，则允许进行使用该资源。进程将信号量减1。

（3） 若此信号量为0，则该资源目前不可用，进程进入睡眠状态，直至信号量值大于0，进程被唤醒，转入步骤（1）。

（4） 当进程不再使用一个信号量控制的资源时，信号量值加1。如果此时有进程正在睡眠等待此信号量，则唤醒此进程。

维护信号量状态的是Linux内核操作系统而不是用户进程。我们可以从头文件/usr/src/linux/include/linux/sem.h 中看到内核用来维护信号量状态的各个结构的定义。信号量是一个数据集合，用户可以单独使用这一集合的每个元素。要调用的第一个函数是semget，用以获得一个信号量ID。Linux2.6.26下定义的信号量结构体：

```
struct semaphore {
        spinlock_t              lock;
        unsigned int            count;
        struct list_head        wait_list;
};
```

从以上信号量的定义中，可以看到信号量底层使用到了spinlock的锁定机制，这个spinlock主要用来确保对count成员的原子性的操作(count--)和测试(count > 0)。

1.信号量的P操作：

(1).void down(struct semaphore *sem);

(2).int down_interruptible(struct semaphore *sem);

(3).int down_trylock(struct semaphore *sem);

说明：

(1)中的函数根据2.6.26中的代码注释，这个函数已经out了(Use of this function is deprecated)，所以从实用角度，彻底忘了它吧。

(2)最常用，函数原型

```
/**
* 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)
{
        unsigned long flags;
        int result = 0;

        spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                result =
__down_interruptible(sem);
        spin_unlock_irqrestore(&sem->lock,
flags);

        return result;
}
```

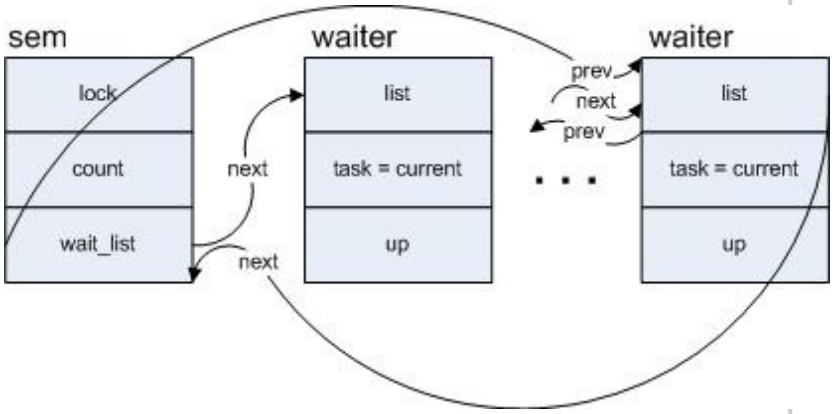
对此函数的理解：在保证原子操作的前提下，先测试count是否大于0，如果是说明可以获得信号量，这种情况下需要先将count--，以确保别的进程能否获得该信号量，然后函数返回，

其调用者开始进入临界区。如果没有获得信号量，当前进程利用struct semaphore 中wait_list加入等待队列，开始睡眠。

对于需要休眠的情况，在__down_interruptible()函数中，会构造一个struct semaphore_waiter类型的变量（ struct semaphore_waiter定义如下：

```
struct semaphore_waiter
{
        struct list_head list;
        struct task_struct *task;
        int up;
};
```

），将当前进程赋给task，并利用其list成员将该变量的节点加入到以sem中的wait_list为头部的一个列表中，假设有多个进程在sem上调用down_interruptible，则sem的wait_list上形成的队列如下图：



(注：将一个进程阻塞，一般的经过是先把进程放到等待队列中，接着改变进程的状态，比如设为TASK_INTERRUPTIBLE，然后调用调度函数schedule()，后者将会把当前进程从cpu的运行队列中摘下)

(3)试图去获得一个信号量，如果没有获得，函数立刻返回1而不会让当前进程进入睡眠状态。

2.信号量的V操作

void up(struct semaphore *sem)；

原型如下：

```
/**
* 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)
{
        unsigned long flags;

        spin_lock_irqsave(&sem->lock, flags);
        if (likely(list_empty(&sem->wait_list)))
                sem->count++;
        else
                __up(sem);
        spin_unlock_irqrestore(&sem->lock,
flags);
}
```

如果没有其他线程等待在目前即将释放的信号量上，那么只需将count++即可。如果有其他线程正因为等待该信号量而睡眠，那么调用__up.

__up的定义：

```
static noinline void __sched __up(struct
semaphore *sem)
{
        struct semaphore_waiter *waiter =
list_first_entry(&sem->wait_list,    struct
semaphore_waiter, list);
        list_del(&waiter->list);
        waiter->up = 1;
        wake_up_process(waiter->task);
}
```

这个函数首先获得sem所在的wait_list为头部的链表的第一个有效节点，然后从链表中将其删除，然后唤醒该节点上睡眠的进程。

由此可见，对于sem上的每次down_interruptible调用，都会在sem的wait_list链表尾部加入一新的节点。对于sem上的每

次up调用，都会删除掉wait_list链表中的第一个有效节点，并唤醒睡眠在该节点上的进程。

关于Linux环境下信号量其他API 详见LKD和ULD

二、互斥体

互斥体实现了"互相排斥"（mutual exclusion）同步的简单形式（所以名为互斥体(mutex)）。互斥体禁止多个线程同时进入受保护的代码"临界区"（critical section）。因此，在任意时刻，只有一个线程被允许进入这样的代码保护区。

任何线程在进入临界区之前，必须获取（acquire）与此区域相关联的互斥体的所有权。如果已有另一线程拥有了临界区的互斥体，其他线程就不能再进入其中。这些线程必须等待，直到当前的属主线程释放（release）该互斥体。

什么时候需要使用互斥体呢？互斥体用于保护共享的易变代码，也就是，全局或静态数据。这样的数据必须通过互斥体进行保护，以防止它们在多个线程同时访问时损坏

Linux 2.6.26中mutex的定义：

```
struct mutex {
        /* 1: unlocked, 0: locked, negative:
locked, possible waiters */
        atomic_t                count;
        spinlock_t              wait_lock;
        struct list_head        wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
        struct thread_info      *owner;
        const char              *name;
        void                    *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map       dep_map;
#endif
};
```

对比前面的struct semaphore，struct mutex除了增加了几个作为debug用途的成员变量外，和semaphore几乎长得一样。但是mutex的引入主要是为了提供互斥机制，以避免多个进程同时在一个临界区中运行。

如果静态声明一个count=1的semaphore变量，可以使用DECLARE_MUTEX(name)，DECLARE_MUTEX(name)实际上是定义一个semaphore，所以它的使用应该对应信号量的P,V函数.

如果要定义一个静态mutex型变量，应该使用DEFINE_MUTEX

如果在程序运行期要初始化一个mutex变量，可以使用mutex_init（mutex），mutex_init是个宏，在该宏定义的内部，会调用__mutex_init函数。

```
#define mutex_init(mutex)
\
do {
\
        static struct lock_class_key __key;
\


\
        __mutex_init((mutex), #mutex, &__key);
\
} while (0)
```

__mutex_init定义如下：

```
/***
* mutex_init - initialize the mutex
* @lock: the mutex to be initialized
*
* Initialize the mutex to unlocked state.
*
* It is not allowed to initialize an already
locked mutex.
*/
void
__mutex_init(struct mutex *lock, const char
*name, struct lock_class_key *key)
{
        atomic_set(&lock->count, 1);
        spin_lock_init(&lock->wait_lock);
        INIT_LIST_HEAD(&lock->wait_list);

        debug_mutex_init(lock, name, key);
}
```

从__mutex_init的定义可以看出，在使用mutex_init宏来初始化一个mutex变量时，应该使用mutex的指针型。

mutex上的P,V操作：void mutex_lock(struct mutex *lock)和void __sched mutex_unlock(struct mutex *lock)

从原理上讲，mutex实际上是count=1情况下的semaphore，所以其PV操作应该和semaphore是一样的。但是在实际的Linux代码上，出于性能优化的角度，并非只是单纯的重用down_interruptible和up的代码。以ARM平台的mutex_lock为例，实际上是将mutex_lock分成两部分实现：fast path和slow path，主要是基于这样一个事实：在绝大多数情况下，试图获得互斥体的代码总是可以成功获得。所以Linux的代码针对这一事实用ARM V6上的LDREX和STREX指令来实现fast path以期获得最佳的执行性能。这里对于mutex的实现细节，不再多说，如欲深入了解，参考APUE和ULD

三、自旋锁

自旋锁它是为为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是**互斥锁**，还是**自旋锁**，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是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，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

自旋锁一般原理

跟互斥锁一样，一个执行单元要想访问被自旋锁保护的**共享资源**，必须先得到锁，在访问完共享资源后，必须释放锁。如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取自旋锁时锁已经有保持者，那么获取锁操作将自旋在那里，直到该自旋锁的保持者释放了锁。由此我们可以看出，自旋锁是一种比较低级的保护数据结构或代码片段的原始方式，这种锁可能存在两个问题：死锁和过多占用cpu资源。

自旋锁适用情况

自旋锁比较适用于锁使用者保持锁时间比较短的情况。正是由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，**自旋锁的效率远高于互斥锁**。信号量和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因此**只能在进程上下文使用**，而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。如果被保护的共享资源只在进程上下文访问，使用信号量保护该共享资源非常合适，如果对共享资源的访问时间非常短，自旋锁也可以。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。自旋锁保持期间是抢占失效的，而信号量和读写信号量保持期间是可以被抢占的。自旋锁只有在内核可抢占或SMP（多处理器）的情况下才真正需要，在单CPU且不可抢占的内核下，自旋锁的所有操作都是空操作。另外格外注意一点：**自旋锁不能递归使用**。

关于自旋锁的定义以及相应的API

自旋锁定义： linux/Spinlock.h

```
typedef struct spinlock {
          union { //联合
             struct raw_spinlock rlock;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct
raw_spinlock, dep_map))
```

```
                struct{
                        u8 __padding[LOCK_PADSIZE];
                        struct lockdep_map dep_map;
                };
#endif
        };
} spinlock_t;
```

定义和初始化

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
void spin_lock_init(spinlock_t *lock);
```

自旋锁操作：

```
//加锁一个自旋锁函数
void spin_lock(spinlock_t *lock);
//获取指定的自旋锁
void spin_lock_irq(spinlock_t *lock);
//禁止本地中断获取指定的锁
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);      //保存本地中断的状态,禁止本地中断,并获取指定的锁
void spin_lock_bh(spinlock_t *lock)
//安全地避免死锁， 而仍然允许硬件中断被服务


//释放一个自旋锁函数
void spin_unlock(spinlock_t *lock);
//释放指定的锁
void spin_unlock_irq(spinlock_t *lock);
//释放指定的锁,并激活本地中断
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags); //释放指定的锁,并让本地中断恢复到以前的状态
void spin_unlock_bh(spinlock_t *lock);
//对应于spin_lock_bh


//非阻塞锁
int spin_trylock(spinlock_t *lock);
//试图获得某个特定的自旋锁,如果该锁已经被争用,该方法会立刻返回一个非0值,
```

```
//而不会自旋等待锁被释放,如果成果获得了这个锁,那么就返回0.
int spin_trylock_bh(spinlock_t *lock);
//这些函数成功时返回非零( 获得了锁 ), 否则 0. 没
有"try"版本来禁止中断.

//其他
int spin_is_locked(spinlock_t *lock);
//和try_lock()差不多
```

四、信号量、互斥体和自旋锁的区别

信号量/互斥体和自旋锁的区别

信号量/互斥体允许进程睡眠属于睡眠锁，自旋锁则不允许调用者睡眠，而是让其循环等待，所以有以下区别应用

1）、信号量和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因而自旋锁适合于保持时间非常短的情况

2）、自旋锁可以用于中断，不能用于进程上下文(会引起死锁)。而信号量不允许使用在中断中，而可以用于进程上下文

3）、自旋锁保持期间是抢占失效的，自旋锁被持有时，内核不能被抢占，而信号量和读写信号量保持期间是可以被抢占的

另外需要注意的是

1）、信号量锁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，另一进程企图获取本自旋锁，死锁就会发生。

2）、在你占用信号量的同时不能占用自旋锁，因为在你等待信号量时可能会睡眠，而在持有自旋锁时是不允许睡眠的。

信号量和互斥体之间的区别

概念上的区别：

信号量：是进程间（线程间）同步用的，一个进程（线程）完成了某一个动作就通过信号量告诉别的进程（线程），别的进程（线程）再进行某些动作。有二值和多值信号量之分。

互斥锁：是线程间互斥用的，一个线程占用了某一个共享资源，那么别的线程就无法访问，直到这个线程离开，其他的线程才开始可以使用这个共享资源。可以把互斥锁看成二值信号量。

上锁时：

信号量: 只要信号量的value大于0，其他线程就可以sem_wait成功，成功后信号量的value减一。若value值不大于0，则sem_wait阻塞，直到sem_post释放后value值加一。一句话，信号量的value>=0。

互斥锁: 只要被锁住，其他任何线程都不可以访问被保护的资源。如果没有锁，获得资源成功，否则进行阻塞等待资源可用。一句话，线程互斥锁的vlaue可以为负数。

使用场所：

信号量主要适用于进程间通信，当然，也可用于线程间通信。而互斥锁只能用于线程间通信。

分类: OS/Linux

« 上一篇：Linux写时拷贝技术(copy-on-write)
» 下一篇：函数指针和指针函数

posted on 2012-07-21 14:50 as_ 阅读(25478) 评论(4) 编辑 收藏