title: 【TencentOS tiny】深度源码分析（6）——互斥锁 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-12 23:17:54 img: coverImg: password: summary: tags: - TencentOS tiny - RTOS - 操作系统 - 物联网 categories: - 操作系统 - TencentOS tiny

互斥锁

互斥锁又称互斥互斥锁，是一种特殊的信号量，它和信号量不同的是，它具有互斥锁所有权、递归访问以及优先级继承等特性，在操作系统中常用于对临界资源的独占式处理。在任意时刻互斥锁的状态只有两种，开锁或闭锁，当互斥锁被任务持有时，该互斥锁处于闭锁状态，当该任务释放互斥锁时，该互斥锁处于开锁状态。

- 一个任务持有互斥锁就表示它拥有互斥锁的所有权，只有该任务才能释放互斥锁，同时其他任务将不能持有该互斥锁，这就是互斥锁的所有权特性。
- 当持有互斥锁的任务再次获取互斥锁时不会被挂起，而是能递归获取，这就是互斥锁的递归访问特性。这个特性与一般的信号量有很大的不同，在信号量中，由于已经不存在可用的信号量，任务递归获取信号量时会发生挂起任务最终形成死锁。
- 互斥锁还拥有优先级继承机制，它可以将低优先级任务的优先级临时提升到与获取互斥锁的高优先级任务的优先级相同，尽可能降低优先级翻转的危害。

在实际应用中，如果想要实现同步功能，可以使用信号量，虽然互斥锁也可以用于任务与任务间的同步，但互斥锁更多的是用于临界资源的互斥访问。

使用互斥锁对临界资源保护时，任务必须先获取互斥锁以获得访问该资源的所有权，当任务使用资源后，必须释放互斥锁以便其他任务可以访问该资源（而使用信号量保护临界资源时则可能发生优先级翻转，且危害是不可控的）。

优先级翻转

简单来说就是高优先级任务在等待低优先级任务执行完毕，这已经违背了操作系统的设计初衷（抢占式调度）。

为什么会发生优先级翻转？

当系统中某个临界资源受到一个互斥锁保护时，任务访问该资源时需要获得互斥锁，如果这个资源正在被一个低优先级任务使用，此时互斥锁处于闭锁状态；如果此时一个高优先级任务想要访问该资源，那么高优先级任务会因为获取不到互斥锁而进入阻塞态，此时就形成高优先级任务在等待低优先级任务运行（等待它释放互斥锁）。

优先级翻转是会产生危害的，在一开始设计系统的时候，就已经指定任务的优先级，越重要的任务优先级越高，但是发生优先级翻转后，高优先级任务在等待低优先级任务，这就有可能让高优先级任务得不到有效的处理，严重时可能导致系统崩溃。

优先级翻转的危害是不可控的，因为低优先级任务很可能会被系统中其他中间优先级任务（低优先级与高优先级任务之间的优先级任务）抢占，这就有可能导致高优先级任务将等待所有中间优先级任务运行完毕的情况，这种情况对高优先级任务来说是不可接受的，也是违背了操作系统设计的原则。

优先级继承

在TencentOS tiny 中为了降低优先级翻转产生的危害使用了优先级继承算法：暂时提高占有某种临界资源的低优先级任务的优先级，使之与在所有等待该资源的任务中，优先级最高的任务优先级相等，而当这个低优先级任务执行完毕释放该资源时，优先级恢复初始设定值（此处可以看作是低优先级任务临时继承了高优先级任务的优先级）。因此，继承优先级的任务避免了系统资源被任何中间优先级任务抢占。互斥锁的优先级继承机制，它确保高优先级任务进入阻塞状态的时间尽可能短，以及将已经出现的“优先级翻转”危害降低到最小，但不能消除优先级翻转带来的危害。

值得一提的是TencentOS tiny 在持有互斥锁时还允许调用修改任务优先级的API接口。

互斥锁的数据结构

互斥锁控制块

TencentOS tiny 通过互斥锁控制块操作互斥锁，其数据类型为k_mutex_t，互斥锁控制块由多个元素组成。

- pend_obj有点类似于面向对象的继承，继承一些属性，里面有描述内核资源的类型（如互斥锁、队列、互斥量等，同时还有一个等待列表list）。
- pend_nesting实际上是一个uint8_t类型的变量，记录互斥锁被获取的次数，如果pend_nesting为0则表示开锁状态，不为0则表示闭锁状态，且它的值记录了互斥量被获取的次数（拥有递归访问特性）
- *owner是任务控制块指针，记录当前互斥锁被哪个任务持有。
- owner_orig_prio变量记录了持有互斥锁任务的优先级，因为有可能发生优先级继承机制而临时改变任务的优先级。（拥有优先级继承机制）。
- owner_list是一个列表节点，当互斥锁被任务获取时，该节点会被添加到任务控制块的task->mutex_own_list列表中，表示任务自己获取到的互斥锁有哪些。当互斥锁被完全释放时（pend_nesting等于0），该节点将从任务控制块的task->mutex_own_list列表中移除。
- prio_pending变量比较有意思：在一般的操作系统中，任务在持有互斥锁时不允许修改优先级，但在TencentOS tiny 中是允许的，就是因为这个变量，当一个任务处于互斥锁优先级反转的时候，我假设他因为优先级反转优先级得到了提升，此时有用户企图改变这个任务的优先级，但这个更改后的优先级会使此任务违背其优先级必须比所有等待他所持有的互斥锁的任务还高的原则时，此时需要将用户的这次优先级更改请求记录到prio_pending里，待其释放其所持有的互斥锁后再更改，相当于将任务优先级的更改延后了。

> 举个例子：好比一个任务优先级是10，且持有一把锁，此时一个优先级为6的任务尝试获取这把锁，那么此任务优先级会被提升为6，如果此时用户试图更改他的优先级为7，那么不能立刻响应这次请求，必须要等这把锁放掉的时候才能做真正的优先级修改

```
typedef struct k_mutex_st {
    pend_obj_t      pend_obj;
    k_nesting_t     pend_nesting;
    k_task_t       *owner;
    k_prio_t        owner_orig_prio;
    k_list_t        owner_list;
} k_mutex_t;
```

```
typedef struct k_task_st {
#if TOS_CFG_MUTEX_EN > 0u
    k_list_t            mutex_own_list;     /**< 任务拥有的互斥量 */
    k_prio_t            prio_pending;       /*< 在持有互斥锁时修改任务优先级将被记录
到这个变量中，在释放持有的互斥锁时再更改 */
#endif
} k_task_t;
```

与互斥锁相关的宏定义

在tos_config.h中，使能互斥锁的宏定义是TOS_CFG_MUTEX_EN

```
#define TOS_CFG_MUTEX_EN            1u
```

创建互斥锁

系统中每个互斥锁都有对应的互斥锁控制块，互斥锁控制块中包含了互斥锁的所有信息，比如它的等待列表、它的资源类型，以及它的互斥锁值，那么可以想象一下，创建互斥锁的本质是不是就是对互斥锁控制块进行初始化呢？很显然就是这样子的。因为在后续对互斥锁的操作都是通过互斥锁控制块来操作的，如果控制块没有信息，那怎么能操作嘛~

创建互斥锁函数是tos_mutex_create()，传入一个互斥锁控制块的指针*mutex即可。

互斥锁的创建实际上就是调用pend_object_init()函数将互斥锁控制块中的mutex->pend_obj成员变量进行初始化，它的资源类型被标识为PEND_TYPE_MUTEX。然后将mutex->pend_nesting成员变量设置为0，表示互斥锁处于开锁状态；将mutex->owner成员变量设置为K_NULL，表示此事并无任务持有互斥锁；将mutex->owner_orig_prio成员变量设置为默认值K_TASK_PRIO_INVALID，毕竟此事没有任务持有互斥锁，也无需记录持有互斥锁任务的优先级。最终调用tos_list_init()函数将互斥锁的持有互斥锁任务的列表节点owner_list。

```
__API__ k_err_t tos_mutex_create(k_mutex_t *mutex)
{
    TOS_PTR_SANITY_CHECK(mutex);

    pend_object_init(&mutex->pend_obj, PEND_TYPE_MUTEX);
    mutex->pend_nesting     = (k_nesting_t)0u;
    mutex->owner            = K_NULL;
    mutex->owner_orig_prio  = K_TASK_PRIO_INVALID;
    tos_list_init(&mutex->owner_list);

    return K_ERR_NONE;
}
```

销毁互斥锁

互斥锁销毁函数是根据互斥锁控制块直接销毁的，销毁之后互斥锁的所有信息都会被清除，而且不能再次使用这个互斥锁，当互斥锁被销毁时，其等待列表中存在任务，系统有必要将这些等待这些任务唤醒，并告知任务互斥锁已经被销毁了PEND_STATE_DESTROY。然后产生一次任务调度以切换到最高优先级任务执行。

TencentOS tiny 对互斥锁销毁的处理流程如下：

1. 调用pend_is_nopending()函数判断一下是否有任务在等待互斥锁
2. 如果有则调用pend_wakeup_all()函数将这些任务唤醒，并且告知等待任务互斥锁已经被销毁了（即设置任务控制块中的等待状态成员变量pend_state为PEND_STATE_DESTROY）。
3. 调用pend_object_deinit()函数将互斥锁控制块中的内容清除，最主要的是将控制块中的资源类型设置为PEND_TYPE_NONE，这样子就无法使用这个互斥锁了。
4. 将mutex->pend_nesting成员变量恢复为默认值0。
5. 如果删除的时候有任务持有互斥锁，那么调用mutex_old_owner_release()函数将互斥锁释放。
6. 进行任务调度knl_sched()

注意：如果互斥锁控制块的RAM是由编译器静态分配的，所以即使是销毁了互斥锁，这个内存也是没办法释放的。当然你也可以使用动态内存为互斥锁控制块分配内存，只不过在销毁后要将这个内存释放掉，避免内存泄漏。

```
__API__ k_err_t tos_mutex_destroy(k_mutex_t *mutex)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_PTR_SANITY_CHECK(mutex);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&mutex->pend_obj, PEND_TYPE_MUTEX)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    TOS_CPU_INT_DISABLE();

    if (!pend_is_nopending(&mutex->pend_obj)) {
        pend_wakeup_all(&mutex->pend_obj, PEND_STATE_DESTROY);
    }

    pend_object_deinit(&mutex->pend_obj);
    mutex->pend_nesting = (k_nesting_t)0u;

    if (mutex->owner) {
        mutex_old_owner_release(mutex);
    }

    TOS_CPU_INT_ENABLE();
    knl_sched();
```

```
    return K_ERR_NONE;
}
```

获取互斥锁

tos_mutex_pend_timed()函数用于获取互斥锁，互斥锁就像是临界资源的令牌，任务只有获取到互斥锁时才能访问临界资源。当且仅当互斥锁处于开锁的状态，任务才能获取互斥锁成功，当任务持有了某个互斥锁的时候，其它任务就无法获取这个互斥锁，需要等到持有互斥锁的任务进行释放后，其他任务才能获取成功，任务通过互斥锁获取函数来获取互斥锁的所有权，任务对互斥锁的所有权是独占的，任意时刻互斥锁只能被一个任务持有，如果互斥锁处于开锁状态，那么获取该互斥锁的任务将成功获得该互斥锁，并拥有互斥锁的使用权；如果互斥锁处于闭锁状态，获取该互斥锁的任务将无法获得互斥锁，任务将可能被阻塞，也可能立即返回，阻塞时间timeout由用户指定，在指定时间还无法获取到互斥锁时，将发送超时，等待任务将自动恢复为就绪态。在任务被阻塞之前，会进行优先级继承，如果当前任务优先级比持有互斥锁的任务优先级高，那么将会临时提升持有互斥锁任务的优先级。

TencentOS tiny 提供两组API接口用于获取互斥锁，分别是tos_mutex_pend_timed()和tos_mutex_pend()，主要的差别是参数不同：可选阻塞与永久阻塞获取互斥锁，实际获取的过程都是一样的。获取互斥锁的过程如下：

1. 首先检测传入的参数是否正确，此处不仅会检查互斥锁控制块的信息，还会调用TOS_IN_IRQ_CHECK()检查上下文是否处于中断中，因为互斥锁的操作是不允许在中断中进行的。
2. 判断互斥锁控制块中的mutex->pend_nesting成员变量是否为0，为0表示互斥锁处于开锁状态，调用mutex_fresh_owner_mark()函数将获取互斥锁任务的相关信息保存到互斥锁控制块中，如mutex->pend_nesting成员变量的值变为1表示互斥锁处于闭锁状态，其他任务无法获取，mutex->owner成员变量指向当前获取互斥锁的任务控制块，mutex->owner_orig_prio成员变量则是记录当前任务的优先级，最终使用tos_list_add()函数将互斥锁控制块的mutex->owner_list节点挂载到任务控制块的task->mutex_own_list列表中，任务获取成功后返回K_ERR_NONE。
3. 如果互斥锁控制块中的mutex->pend_nesting成员变量不为0，则表示互斥锁处于闭锁状态，那么由于互斥锁具有递归访问特性，那么会判断一下是不是已经持有互斥锁的任务再次获取互斥锁（knl_is_self()），因为这也是允许的，判断一下mutex->pend_nesting 成员变量的值是否为(k_nesting_t)-1，如果是则表示递归访问次数达到最大值，互斥锁已经溢出了，返回错误代码K_ERR_MUTEX_NESTING_OVERFLOW。否则就将mutex->pend_nesting成员变量的值加1，返回K_ERR_MUTEX_NESTING表示递归获取成功。
4. 如果互斥锁处于闭锁状态，且当前任务并未持有互斥锁，看一下用户指定的阻塞时间timeout是否为不阻塞TOS_TIME_NOWAIT，如果不阻塞则直接返回K_ERR_PEND_NOWAIT错误代码。
5. 如果调度器被锁了knl_is_sched_locked()，则无法进行等待操作，返回错误代码K_ERR_PEND_SCHED_LOCKED，毕竟需要切换任务，调度器被锁则无法切换任务。
6. 最最最最重要的特性来了，在阻塞当前任务之前，需要判断一下当前任务与持有互斥锁的任务优先级大小情况，如果当前任务优先级比持有互斥锁任务优先级大，则需要进行优先级继承，临时将持有互斥锁任务的优先级提升到当前优先级，通过tos_task_prio_change()函数进行改变优先级。
7. 调用pend_task_block()函数将任务阻塞，该函数实际上就是将任务从就绪列表中移除k_rdyq.task_list_head[task_prio]，并且插入到等待列表中object->list，如果等待的时间不是永久等待TOS_TIME_FOREVER，还会将任务插入时间列表中k_tick_list，阻塞时间为timeout，然后进行一次任务调度knl_sched()。
8. 当程序能行到pend_state2errno()时，则表示任务等获取到互斥锁，又或者等待发生了超时，那么就调用pend_state2errno()函数获取一下任务的等待状态，看一下是哪种情况导致任务恢复运行，如果

任务已经获取到互斥锁，那么需要调用mutex_new_owner_mark()函数标记一下获取任务的信息，将获取互斥锁任务的相关信息保存到互斥锁控制块中。

注意：当获取互斥锁的任务能从阻塞中恢复运行，也不一定是获取到互斥锁，也可能是发生了超时，因此在写程序的时候必须要判断一下获取的互斥锁状态，如果返回值是K_ERR_NONE与K_ERR_MUTEX_NESTING则表示获取成功！

```
__API__ k_err_t tos_mutex_pend_timed(k_mutex_t *mutex, k_tick_t timeout)
{
    TOS_CPU_CPSR_ALLOC();
    k_err_t err;

    TOS_PTR_SANITY_CHECK(mutex);
    TOS_IN_IRQ_CHECK();

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&mutex->pend_obj, PEND_TYPE_MUTEX)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    TOS_CPU_INT_DISABLE();
    if (mutex->pend_nesting == (k_nesting_t)0u) { // first come
        mutex_fresh_owner_mark(mutex, k_curr_task);
        TOS_CPU_INT_ENABLE();
        return K_ERR_NONE;
    }

    if (knl_is_self(mutex->owner)) { // come again
        if (mutex->pend_nesting == (k_nesting_t)-1) {
            TOS_CPU_INT_ENABLE();
            return K_ERR_MUTEX_NESTING_OVERFLOW;
        }
        ++mutex->pend_nesting;
        TOS_CPU_INT_ENABLE();
        return K_ERR_MUTEX_NESTING;
    }

    if (timeout == TOS_TIME_NOWAIT) { // no wait, return immediately
        TOS_CPU_INT_ENABLE();
        return K_ERR_PEND_NOWAIT;
    }

    if (knl_is_sched_locked()) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_PEND_SCHED_LOCKED;
    }

    if (mutex->owner->prio > k_curr_task->prio) {
        // PRIORITY INVERSION:
        // we are declaring a mutex, which's owner has a lower(numerically bigger)
priority.
```

```
            // make owner the same priority with us.
            tos_task_prio_change(mutex->owner, k_curr_task->prio);
        }

        pend_task_block(k_curr_task, &mutex->pend_obj, timeout);

        TOS_CPU_INT_ENABLE();
        knl_sched();

        err = pend_state2errno(k_curr_task->pend_state);

        if (err == K_ERR_NONE) {
            // good, we are the owner now.
            TOS_CPU_INT_DISABLE();
            mutex_new_owner_mark(mutex, k_curr_task);
            TOS_CPU_INT_ENABLE();
        }

        return err;
    }

    __API__ k_err_t tos_mutex_pend(k_mutex_t *mutex)
    {
        TOS_PTR_SANITY_CHECK(mutex);

        return tos_mutex_pend_timed(mutex, TOS_TIME_FOREVER);
    }
```

mutex_fresh_owner_mark与mutex_new_owner_mark()函数的实现：

```
    __STATIC_INLINE__ void mutex_fresh_owner_mark(k_mutex_t *mutex, k_task_t *task)
    {
        mutex->pend_nesting     = (k_nesting_t)1u;
        mutex->owner            = task;
        mutex->owner_orig_prio  = task->prio;

        tos_list_add(&mutex->owner_list, &task->mutex_own_list);
    }

    __STATIC_INLINE__ void mutex_new_owner_mark(k_mutex_t *mutex, k_task_t *task)
    {
        k_prio_t highest_pending_prio;

        mutex_fresh_owner_mark(mutex, task);

        // we own the mutex now, make sure our priority is higher than any one in the
    pend list.
        highest_pending_prio = pend_highest_prio_get(&mutex->pend_obj);
        if (task->prio > highest_pending_prio) {
            tos_task_prio_change(task, highest_pending_prio);
        }
    }
```

释放互斥锁

互斥锁的释放是不允许在中断中释放的，主要的原因是因为中断中没有上下文的概念，所以中断上下文不能持有，也不能释放互斥锁；互斥锁有所属关系，只有持有互斥锁的任务才能将互斥锁释放，而持有者是任务。

任务想要访问某个资源的时候，需要先获取互斥锁，然后进行资源访问，在任务使用完该资源的时候，必须要及时释放互斥锁，这样别的任务才能访问临界资源。任务可以调用tos_mutex_post()函数进行释放互斥锁，当互斥锁处于开锁状态时就表示我已经用完了，别人可以获取互斥锁以访问临界资源。

使用tos_mutex_post()函数接口时，只有已持有互斥锁所有权的任务才能释放它，当任务调用tos_mutex_post()函数时会将互斥锁释放一次，当互斥锁完全释放完毕（mutex->pend_nesting成员变量的值为0）就变为开锁状态，等待获取该互斥锁的任务将被唤醒。如果任务的优先级被互斥锁的优先级翻转机制临时提升，那么当互斥锁被释放后，任务的优先级将恢复为原本设定的优先级。

TencentOS tiny 中可以只让等待中的一个任务获取到互斥锁（在等待的任务中具有最高优先级）。

在tos_mutex_post()函数中的处理也是非常简单明了的，其执行思路如下：

1. 首先进行传入的互斥锁控制块相关的参数检测，然后判断一下是否是持有互斥锁的任务来释放互斥锁，如果是则进行释放操作，如果不是则返回错误代码K_ERR_MUTEX_NOT_OWNER。
2. 将mutex->pend_nesting成员变量的值减1，然后判断它的值是否为0，如果不为0则表示当前任务还是持有互斥锁的，也无需进行后续的操作，直接返回K_ERR_MUTEX_NESTING。
3. 如果mutex->pend_nesting成员变量的值为0，则表示互斥锁处于开锁状态，则需要调用mutex_old_owner_release()函数完全释放掉互斥锁，在这个函数中会将互斥锁控制块的成员变量（如owner_list、owner、owner_orig_prio等都设置为初始值），此外还有最重要的就是判断一下任务是否发生过优先级继承，如果是则需要将任务恢复为原本的优先级，否则就无效理会。
4. 调用pend_is_nopending()函数判断一下是否有任务在等待互斥锁，如果没有则返回K_ERR_NONE表示释放互斥锁成功，因为此时没有唤醒任务也就无需任务调度，直接返回即可。
5. 如果有任务在等待互斥锁，则直接调用pend_wakeup_one()函数唤醒一个等待任务，这个任务在等待任务中是处于最高优先级的，因为TencentOS tiny 的等待任务是按照优先级进行排序。
6. 进行一次任务调度knl_sched()。

```
__API__ k_err_t tos_mutex_post(k_mutex_t *mutex)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_PTR_SANITY_CHECK(mutex);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&mutex->pend_obj, PEND_TYPE_MUTEX)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    TOS_CPU_INT_DISABLE();
    if (!knl_is_self(mutex->owner)) {
        TOS_CPU_INT_ENABLE();
```

```
        return K_ERR_MUTEX_NOT_OWNER;
    }

    if (mutex->pend_nesting == (k_nesting_t)0u) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_MUTEX_NESTING_OVERFLOW;
    }

    --mutex->pend_nesting;
    if (mutex->pend_nesting > (k_nesting_t)0u) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_MUTEX_NESTING;
    }

    mutex_old_owner_release(mutex);

    if (pend_is_nopending(&mutex->pend_obj)) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_NONE;
    }

    pend_wakeup_one(&mutex->pend_obj, PEND_STATE_POST);
    TOS_CPU_INT_ENABLE();
    knl_sched();

    return K_ERR_NONE;
}
```

持有互斥锁的任务释放互斥锁`mutex_old_owner_release()`。

```
__STATIC_INLINE__ void mutex_old_owner_release(k_mutex_t *mutex)
{
    k_task_t *owner;

    owner = mutex->owner;

    tos_list_del(&mutex->owner_list);
    mutex->owner = K_NULL;

    // the right time comes! let's do it!
    if (owner->prio_pending != K_TASK_PRIO_INVALID) {
        tos_task_prio_change(owner, owner->prio_pending);
        owner->prio_pending = K_TASK_PRIO_INVALID;
    } else if (owner->prio != mutex->owner_orig_prio) {
        tos_task_prio_change(owner, mutex->owner_orig_prio);
        mutex->owner_orig_prio = K_TASK_PRIO_INVALID;
    }
}
```

喜欢就关注我吧！

相关代码可以在公众号后台回复“19”获取。