

title: 【TencentOS tiny】深度源码分析（5）——信号量 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-12 23:16:54 img: coverImg: password: summary: tags: - TencentOS tiny - RTOS - 操作系统 - 物联网 categories: - 操作系统 - TencentOS tiny

信号量

信号量（sem）在操作系统中是一种实现系统中任务与任务、任务与中断间同步或者临界资源互斥保护的机制。在多任务系统中，各任务之间常需要同步或互斥，信号量就可以为用户提供这方面的支持。

抽象来说，信号量是一个非负整数，每当信号量被获取（pend）时，该整数会减一，当该整数的值为0时，表示信号量处于无效状态，将无法被再次获取，所有试图获取它的任务将进入阻塞态。通常一个信号量是有计数值的，它的计数值可以用于系统资源计数（统计）。

一般来说信号量的值有两种：

- 0：表示没有积累下来的post信号量操作，且可能有任务阻塞在此信号量上。
- 正值：表示有一个或多个post信号量操作。

一般来说信号量多用于同步而非互斥，因为操作系统中会提供另一种互斥机制（互斥锁），互斥量的互斥作用更完善：互斥锁有优先级继承机制，而信号量则没有这个机制，此外互斥量还拥有所有者属性，我们会在后续讲解。

信号量也如队列一样，拥有阻塞机制。任务需要等待某个中断发生后，再去执行对应的处理，那么任务可以处于阻塞态等待信号量，直到中断发生后释放信号量后，该任务才被唤醒去执行对应的处理。在释放（post）信号量的时候能立即将等待的任务转变为就绪态，如果任务的优先级在就绪任务中是最高的，任务就能立即被运行，这就是操作系统中的“实时响应，实时处理”。在操作系统中使用信号量可以提高处理的效率。

信号量的数据结构

信号量控制块

TencentOS tiny 通过信号量控制块操作信号量，其数据类型为k_sem_t，信号量控制块由多个元素组成，主要有pend_obj_t类型的pend_obj以及k_sem_cnt_t类型的count。而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（如信号量、队列、互斥量等，同时还有一个等待列表list）。而count则是一个简单的变量（它是16位的无符号整数），表示信号量的值。

```
typedef struct k_sem_st {
    pend_obj_t    pend_obj;
    k_sem_cnt_t   count;
} k_sem_t;
```

与信号量相关的宏定义

在tos_config.h中，使能信号量的宏定义是TOS_CFG_SEM_EN

```
#define TOS_CFG_SEM_EN 1u
```

信号量实现

TencentOS tiny 中实现信号量非常简单，核心代码仅仅只有125行，可以说是非常少了。

创建信号量

系统中每个信号量都有对应的信号量控制块，信号量控制块中包含了信号量的所有信息，比如它的等待列表、它的资源类型，以及它的信号量值，那么可以想象一下，创建信号量的本质是不是就是对信号量控制块进行初始化呢？很显然就是这样子的。因为在后续对信号量的操作都是通过信号量控制块来操作的，如果控制块没有信息，那怎么能操作嘛~

创建信号量函数是`tos_sem_create()`，传入两个参数，一个是信号量控制块的指针`*sem`，另一个是信号量的初始值`init_count`，该值是非负整数即可，但主要不能超过65535。

实际上就是调用`pend_object_init()`函数将信号量控制块中的`sem->pend_obj`成员变量进行初始化，它的资源类型被标识为`PEND_TYPE_SEM`。然后将`sem->count`成员变量设置为传递进来的信号量的初始值`init_count`。

```
__API__ k_err_t tos_sem_create(k_sem_t *sem, k_sem_cnt_t init_count)
{
    TOS_PTR_SANITY_CHECK(sem);

    pend_object_init(&sem->pend_obj, PEND_TYPE_SEM);
    sem->count = init_count;

    return K_ERR_NONE;
}
```

销毁信号量

信号量销毁函数是根据信号量控制块直接销毁的，销毁之后信号量的所有信息都会被清除，而且不能再次使用这个信号量，当信号量被销毁时，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，并告知任务信号量已经被销毁了`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 对信号量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信号量
2. 如果有则调用`pend_wakeup_all()`函数将这些任务唤醒，并且告知等待任务信号量已经被销毁了（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()`函数将信号量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信号量了。

4. 进行任务调度`kn1_sched()`

注意：如果信号量控制块的RAM是由编译器静态分配的，所以即使是销毁了信号量，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信号量控制块分配内存，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_sem_destroy(k_sem_t *sem)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_PTR_SANITY_CHECK(sem);

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

    TOS_CPU_INT_DISABLE();

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

    pend_object_deinit(&sem->pend_obj);

    TOS_CPU_INT_ENABLE();
    kn1_sched();

    return K_ERR_NONE;
}
```

获取信号量

`tos_sem_pend()`函数用于获取信号量，当信号量有效的时候，任务才能获取信号量。任务获取了某个信号量时，该信号量的可用个数减一，当它为0的时候，获取信号量的任务会进入阻塞态，阻塞时间`timeout`由用户指定，在指定时间还无法获取到信号量时，将发送超时，等待任务将自动恢复为就绪态。

获取信号量的过程如下：

1. 首先检测传入的参数是否正确。
2. 判断信号量控制块中的`count`成员变量是否大于0，大于0表示存在可用信号量，将`count`成员变量的值减1，任务获取成功后返回`K_ERR_NONE`。
3. 如果不存在信号量则可能会阻塞当前获取的任务，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。
4. 如果调度器被锁了`kn1_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。

5. 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`kn1_sched()`。
6. 当程序能行到`pend_state2errno()`时，则表示任务等获取到信号量，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，并且将结果返回给调用获取信号量的任务。

注意：当获取信号量的任务能从阻塞中恢复运行，也不一定是获取到信号量，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信号量状态，如果是`K_ERR_NONE`则表示获取成功！

```
__API__ k_err_t tos_sem_pend(k_sem_t *sem, k_tick_t timeout)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_PTR_SANITY_CHECK(sem);
    TOS_IN_IRQ_CHECK();

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

    TOS_CPU_INT_DISABLE();

    if (sem->count > (k_sem_cnt_t)0u) {
        --sem->count;
        TOS_CPU_INT_ENABLE();
        return K_ERR_NONE;
    }

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

    if (kn1_is_sched_locked()) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_PEND_SCHED_LOCKED;
    }

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

    TOS_CPU_INT_ENABLE();
    kn1_sched();

    return pend_state2errno(k_curr_task->pend_state);
}
```

释放信号量

任务或者中断服务程序都可以释放信号量（post），释放信号量的本质就是将信号量控制块的count成员变量的值加1，表示信号量有效，不过如果有任务在等待这个信号量时，信号量控制块的count成员变量的值是不会改变的，因为要唤醒等待任务，而唤醒等待任务的本质就是等待任务获取到信号量，信号量控制块的count成员变量的值要减1，这一来一回中，信号量控制块的count成员变量的值是不会改变的。

TencentOS tiny 中可以只让等待中的一个任务获取到信号量，也可以让所有等待任务都获取到信号量。分别对应的API是tos_sem_post()与tos_sem_post_all()。顺便提一点，tos_sem_post_all()的设计模式其实是观察者模式，当一个观察的对象改变后，那么所有的观察者都会知道它改变了，具体可以看看《大话设计模式》这本书。

TencentOS tiny 中设计的很好的地方就是简单与低耦合，这两个api接口本质上都是调用sem_do_post()函数去释放信号量，只是通过opt参数不同选择不同的处理方法。

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

1. 首先判断一下信号量是否溢出了，因为一个整数始终都会溢出的，总不能一直释放信号量让count成员变量的值加1吧，因此必须要判断一下是否溢出，如果sem->count 的值为 (k_sem_cnt_t)-1，则表示已经溢出，无法继续释放信号量，返回错误代码K_ERR_SEM_OVERFLOW。
2. 调用pend_is_nopending()函数判断一下是否有任务在等待信号量，如果没有则将count成员变量的值加1，返回K_ERR_NONE表示释放信号量成功，因为此时没有唤醒任务也就无需任务调度，直接返回即可。
3. 如果有任务在等待信号量，则count成员变量的值无需加1，直接调用pend_wakeup唤醒对应的任务即可，唤醒任务则是根据opt参数进行唤醒，可以唤醒等待中的一个任务或者是所有任务。
4. 进行一次任务调度kn1_sched()。

```
__API__ k_err_t tos_sem_post(k_sem_t *sem)
{
    TOS_PTR_SANITY_CHECK(sem);

    return sem_do_post(sem, OPT_POST_ONE);
}

__API__ k_err_t tos_sem_post_all(k_sem_t *sem)
{
    TOS_PTR_SANITY_CHECK(sem);

    return sem_do_post(sem, OPT_POST_ALL);
}

__STATIC__ k_err_t sem_do_post(k_sem_t *sem, opt_post_t opt)
{
    TOS_CPU_CPSR_ALLOC();

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

```

#endif

TOS_CPU_INT_DISABLE();

if (sem->count == (k_sem_cnt_t)-1) {
    TOS_CPU_INT_ENABLE();
    return K_ERR_SEM_OVERFLOW;
}

if (pend_is_nopending(&sem->pend_obj)) {
    ++sem->count;
    TOS_CPU_INT_ENABLE();
    return K_ERR_NONE;
}

pend_wakeup(&sem->pend_obj, PEND_STATE_POST, opt);

TOS_CPU_INT_ENABLE();
knl_sched();

return K_ERR_NONE;
}

```

关于为什么判断`sem->count`是`(k_sem_cnt_t)-1`就代表溢出呢？我在C语言中举了个简单的例子：

```

#include <stdio.h>

int main()
{
    unsigned int a = ~0;
    if(a == (unsigned int)0xFFFFFFFF)
    {
        printf("OK\n");
    }
    if(a == (unsigned int)-1)
    {
        printf("OK\n");
    }

    printf("unsigned int a = %d \n",a);

    return 0;
}

```

输出：

```

OK
OK
unsigned int a = -1

```

总结

代码精悍短小，思想清晰，非常建议深入学习~

喜欢就关注我吧！



相关代码可以在公众号后台回复“19”获取。