(163条消息) ThreadX内核源码分析 - 计数信号量_arm7star的博客-CSDN博客

blog.csdn.net/arm7star/article/details/123442459

1、计数信号量介绍

计数信号量的信号量值不为0，表示信号量可获取，每次获取信号量，信号量计数器的值减1，为0是，信号量不看获取，释放信号量是每次加1；

计数信号量实现与互斥锁类似，一定程度上可以把互斥锁看出计数为1的信号量，只不过互斥锁有动态优先级调整，信号量没有，互斥锁用于临界资源保护，信号量用于生产消费者这类场景(多个消费者、多个生产者)。

2、信号量获取_tx_semaphore_get

获取信号量的代码比较简单，主要是对计数信号量的计数进行判断，如果信号量计数器不为0，就减1然后返回(获取到了信号量)，如果信号量计数器为0(获取不到信号量)，那么设置自己的阻塞状态，设置超时/终止等情况下的清理函数，设置阻塞的信号量，将自己加入到信号量的阻塞队列，挂起自己，内核的挂起函数在挂起当前线程是会选择下一个执行线程并进行线程切换(切换过程当然也会保存当前线程的上下文)。

_tx_semaphore_get实现代码如下：

```
UINT  _tx_semaphore_get(TX_SEMAPHORE *semaphore_ptr, ULONG wait_option)
{

TX_INTERRUPT_SAVE_AREA

TX_THREAD       *thread_ptr;
TX_THREAD       *next_thread;
TX_THREAD       *previous_thread;
UINT            status;


    /* Default the status to TX_SUCCESS.  */
    status =  TX_SUCCESS;

    /* Disable interrupts to get an instance from the semaphore.  */
    TX_DISABLE

#ifdef TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO

    /* Increment the total semaphore get counter.  */
    _tx_semaphore_performance_get_count++;

    /* Increment the number of attempts to get this semaphore.  */
    semaphore_ptr -> tx_semaphore_performance_get_count++;
#endif

    /* If trace is enabled, insert this event into the trace buffer.  */
    TX_TRACE_IN_LINE_INSERT(TX_TRACE_SEMAPHORE_GET, semaphore_ptr, wait_option, semaphore_ptr -> tx_semaphore_count, TX_POINTER_TO_ULONG_CONVERT(&thread_ptr), TX_TRACE_SEMAPHORE_EVENTS)

    /* Log this kernel call.  */
    TX_EL_SEMAPHORE_GET_INSERT

    /* Determine if there is an instance of the semaphore.  */
    if (semaphore_ptr -> tx_semaphore_count != ((ULONG) 0)) // 信号量计数器不为0，可以获取到信号量
    {

        /* Decrement the semaphore count.  */
        semaphore_ptr -> tx_semaphore_count--; // 简单对计数信号量减1即可，然后返回成功(与互斥锁不同，计数信号量没有记录owner)

        /* Restore interrupts.  */
        TX_RESTORE
    }

    /* Determine if the request specifies suspension.  */
    else if (wait_option != TX_NO_WAIT) // 信号量为0，获取不到信号量，并且wait_option不是TX_NO_WAIT，也就是获取信号量是阻塞的
    {

        /* Determine if the preempt disable flag is non-zero.  */
        if (_tx_thread_preempt_disable != ((UINT) 0)) // 如果禁止了抢占，那么不能阻塞获取不到信号量的线程，否则所有其他就绪线程都得不到调度
        {

            /* Restore interrupts.  */
            TX_RESTORE

            /* Suspension is not allowed if the preempt disable flag is non-zero at this point - return error completion.  */
            status =  TX_NO_INSTANCE; // 获取不到信号量，禁止抢占，返回TX_NO_INSTANCE即可
        }
        else // 没有禁止抢占，获取不到信号量的线程可以进入阻塞状态
        {

            /* Prepare for suspension of this thread.  */

#ifdef TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO

            /* Increment the total semaphore suspensions counter.  */
            _tx_semaphore_performance_suspension_count++;

            /* Increment the number of suspensions on this semaphore.  */
            semaphore_ptr -> tx_semaphore_performance_suspension_count++;
#endif

            /* Pickup thread pointer.  */
            TX_THREAD_GET_CURRENT(thread_ptr) // 获取当前线程_tx_thread_current_ptr

            /* Setup cleanup routine pointer.  */
            thread_ptr -> tx_thread_suspend_cleanup =  &(_tx_semaphore_cleanup); // 设置超时清理函数_tx_semaphore_cleanup(与前面文章介绍的获取内存阻塞一样，超时需要通过_tx_semaphore_cleanup唤醒线程并且从等待信号量队列删除线程)

            /* Setup cleanup information, i.e. this semaphore control
               block.  */
            thread_ptr -> tx_thread_suspend_control_block =  (VOID *) semaphore_ptr; // 阻塞在信号量semaphore_ptr上(线程挂在semaphore_ptr等待链表上，_tx_semaphore_cleanup及释放信号量的线程需要通过semaphore_ptr找到等待信号量的线程)

#ifndef TX_NOT_INTERRUPTABLE

            /* Increment the suspension sequence number, which is used to identify
               this suspension event.  */
            thread_ptr -> tx_thread_suspension_sequence++;
#endif

```

```
            /* Setup suspension list.  */
            if (semaphore_ptr -> tx_semaphore_suspended_count == TX_NO_SUSPENSIONS) // 只有当前线程等待信号量，加入tx_semaphore_suspension_list信号量等待链表
            {

                /* No other threads are suspended.  Setup the head pointer and
                   just setup this threads pointers to itself.  */
                semaphore_ptr -> tx_semaphore_suspension_list = thread_ptr;
                thread_ptr -> tx_thread_suspended_next = thread_ptr;
                thread_ptr -> tx_thread_suspended_previous = thread_ptr;
            }
            else // 有多个线程等待信号量，加入tx_semaphore_suspension_list信号量等待链表，添加到末尾
            {

                /* This list is not NULL, add current thread to the end. */
                next_thread =                                   semaphore_ptr -> tx_semaphore_suspension_list;
                thread_ptr -> tx_thread_suspended_next =        next_thread;
                previous_thread =                               next_thread -> tx_thread_suspended_previous;
                thread_ptr -> tx_thread_suspended_previous =    previous_thread;
                previous_thread -> tx_thread_suspended_next =   thread_ptr;
                next_thread -> tx_thread_suspended_previous =   thread_ptr;
            }

            /* Increment the number of suspensions.  */
            semaphore_ptr -> tx_semaphore_suspended_count++; // 等待信号量的线程个数加1

            /* Set the state to suspended.  */
            thread_ptr -> tx_thread_state =    TX_SEMAPHORE_SUSP; // 设置当前线程的状态，等待信号量超时需要判断状态，其他一些唤醒阻塞操作也需要判断状态(一些状态的线程不能被唤醒或者挂起，或者说线程处于不可中断的阻塞状态)

#ifdef TX_NOT_INTERRUPTABLE

            /* Call actual non-interruptable thread suspension routine.  */
            _tx_thread_system_ni_suspend(thread_ptr, wait_option);

            /* Restore interrupts.  */
            TX_RESTORE
#else

            /* Set the suspending flag.  */
            thread_ptr -> tx_thread_suspending =  TX_TRUE; // 设置tx_thread_suspending，线程还没从就绪链表删除

            /* Setup the timeout period.  */
            thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks =  wait_option; // 等待超时时间，如果wait_option不是无限等待，那么_tx_thread_system_suspend挂起线程时就会启动一个定时器，超时调用tx_thread_suspend_cleanup唤醒线程，然后调用_tx_semaphore_cleanup处理信号量超时事件

            /* Temporarily disable preemption.  */
            _tx_thread_preempt_disable++; // 禁止抢占(_tx_thread_system_suspend会对_tx_thread_preempt_disable减1，调用_tx_thread_system_suspend前必须对_tx_thread_preempt_disable加1)

            /* Restore interrupts.  */
            TX_RESTORE // 允许中断(之前对抢占计数器加1了，_tx_thread_system_suspend执行期间除了能处理中断服务程序外，其他线程是不允许被调度执行的，禁止抢占加上允许中断的目的只是避免阻塞中断并让_tx_thread_system_suspend执行完)

            /* Call actual thread suspension routine.  */
            _tx_thread_system_suspend(thread_ptr); // 挂起当前线程
#endif

```

```
            /* Return the completion status.  */
            status =  thread_ptr -> tx_thread_suspend_status; // 等待超时，清理函数会设置tx_thread_suspend_status为超时；别的线程释放信号量会释放给当前线程，tx_thread_suspend_status会设置为成功，当前线程被唤醒后，根据tx_thread_suspend_status即可知道自己是获取到了信号量还是超时了
        }
    }
    else
    {

        /* Restore interrupts.  */
        TX_RESTORE

        /* Immediate return, return error completion.  */
        status =  TX_NO_INSTANCE;
    }

    /* Return completion status.  */
    return(status);
}
```

至于等待信号量超时，在此略过，处理比较简单，与申请内存超时实现基本一样，可以参考：

ThreadX内核源码分析 - 动态内存管理_arm7star的博客-CSDN博客_threadx 内存管理

3、信号量释放_tx_semaphore_put

释放信号量操作也比较简单，主要是对计数信号量的计数器加1，然后检查是否有线程等待信号量，如果没有，对信号量计数器加1，直接返回即可，如果有等待信号量的线程，将信号量给等待链表第一个等待信号量的线程并唤醒该线程即可(不对信号量加1，直接把该信号量给等待线程，等待信号量的线程阻塞唤醒后不会再对信号量减1，可能有的内核会恢复信号量的值，然后唤醒所有等待信号量的线程，让等待信号量的线程去重新抢信号量，ThreadX内核没有这么做!!!)。

_tx_semaphore_put实现代码如下:

```
UINT  _tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr)
{

TX_INTERRUPT_SAVE_AREA

#ifndef TX_DISABLE_NOTIFY_CALLBACKS
VOID            (*semaphore_put_notify)(struct TX_SEMAPHORE_STRUCT *notify_semaphore_ptr);
#endif

TX_THREAD       *thread_ptr;
UINT            suspended_count;
TX_THREAD       *next_thread;
TX_THREAD       *previous_thread;


    /* Disable interrupts to put an instance back to the semaphore.  */
    TX_DISABLE // 关闭中断

#ifdef TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO

    /* Increment the total semaphore put counter.  */
    _tx_semaphore_performance_put_count++;

    /* Increment the number of puts on this semaphore.  */
    semaphore_ptr -> tx_semaphore_performance_put_count++;
#endif

    /* If trace is enabled, insert this event into the trace buffer.  */
    TX_TRACE_IN_LINE_INSERT(TX_TRACE_SEMAPHORE_PUT, semaphore_ptr, semaphore_ptr -> tx_semaphore_count, semaphore_ptr -> tx_semaphore_suspended_count, TX_POINTER_TO_ULONG_CONVERT(&thread_ptr), TX_TRACE_SEMAPHORE_EVENTS)

    /* Log this kernel call.  */
    TX_EL_SEMAPHORE_PUT_INSERT

    /* Pickup the number of suspended threads.  */
    suspended_count =  semaphore_ptr -> tx_semaphore_suspended_count; // 等待信号量的线程数量

    /* Determine if there are any threads suspended on the semaphore.  */
    if (suspended_count == TX_NO_SUSPENSIONS) // 没有线程等待信号量
    {

        /* Increment the semaphore count.  */
        semaphore_ptr -> tx_semaphore_count++; // 信号量计数器加1，恢复信号量计数器即可

#ifndef TX_DISABLE_NOTIFY_CALLBACKS

        /* Pickup the application notify function.  */
        semaphore_put_notify =  semaphore_ptr -> tx_semaphore_put_notify;
#endif

        /* Restore interrupts.  */
        TX_RESTORE

#ifndef TX_DISABLE_NOTIFY_CALLBACKS

        /* Determine if notification is required.  */
        if (semaphore_put_notify != TX_NULL)
        {

            /* Yes, call the appropriate notify callback function.  */
            (semaphore_put_notify)(semaphore_ptr);
        }
```

```
#endif
    }
    else // 有线程等待信号量(此次信号量计数器还没被修改，信号量还没释放)
    {

        /* A thread is suspended on this semaphore.  */

        /* Pickup the pointer to the first suspended thread.  */
        thread_ptr =  semaphore_ptr -> tx_semaphore_suspension_list; // 获取第一个等待信号量的线程(与互斥锁场景不同，继承优先级的互斥锁是让高优先级线程先获取到互斥锁，避免高优先级线程等待低优先级线程的情况)

        /* Remove the suspended thread from the list.  */

        /* See if this is the only suspended thread on the list.  */
        suspended_count--; // 等待信号量的线程数量减1
        if (suspended_count == TX_NO_SUSPENSIONS) // 如果没有其他线程等待信号量，那么情况信号量的等待链表即可
        {

            /* Yes, the only suspended thread.  */

            /* Update the head pointer.  */
            semaphore_ptr -> tx_semaphore_suspension_list =  TX_NULL;
        }
        else // 有其他线程等待信号量，第一个等待信号量的线程从等待链表删除即可(第一个等待信号量的线程thread_ptr即将获取到信号量)
        {

            /* At least one more thread is on the same expiration list.  */

            /* Update the list head pointer.  */
            next_thread = thread_ptr -> tx_thread_suspended_next;
            semaphore_ptr -> tx_semaphore_suspension_list =   next_thread;

            /* Update the links of the adjacent threads.  */
            previous_thread =                               thread_ptr -> tx_thread_suspended_previous;
            next_thread -> tx_thread_suspended_previous =   previous_thread;
            previous_thread -> tx_thread_suspended_next =   next_thread;
        }

        /* Decrement the suspension count.  */
        semaphore_ptr -> tx_semaphore_suspended_count =  suspended_count; // 更新信号量等待线程个数(suspended_count在前面减1了，减thread_ptr)

        /* Prepare for resumption of the first thread.  */

        /* Clear cleanup routine to avoid timeout.  */
        thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL; // 清空thread_ptr的tx_thread_suspend_cleanup(thread_ptr获取到了信号量，不再需要清理函数)

#ifndef TX_DISABLE_NOTIFY_CALLBACKS

        /* Pickup the application notify function.  */
        semaphore_put_notify =  semaphore_ptr -> tx_semaphore_put_notify;
#endif

        /* Put return status into the thread control block.  */
        thread_ptr -> tx_thread_suspend_status =  TX_SUCCESS; // thread_ptr获取到信号量，tx_thread_suspend_status设置为TX_SUCCESS，thread_ptr唤醒后会用tx_thread_suspend_status作为返回值，用于判断是否获取到信号量(如果超时，超时函数会设置tx_thread_suspend_status为超时状态，阻塞的线程唤醒后并不知道自己唤醒的原因，因此需要通过tx_thread_suspend_status来判断自己是怎么被唤醒的)

#ifdef TX_NOT_INTERRUPTABLE

        /* Resume the thread!  */
        _tx_thread_system_ni_resume(thread_ptr);

        /* Restore interrupts.  */
        TX_RESTORE
#else

        /* Temporarily disable preemption.  */
        _tx_thread_preempt_disable++; // 禁止抢占(_tx_thread_system_resume会对抢占计数器减1，这里必须加1)

        /* Restore interrupts.  */
        TX_RESTORE // 允许中断

        /* Resume thread.  */
        _tx_thread_system_resume(thread_ptr); // 唤醒thread_ptr(如果有抢占的话，会发生线程切换)
#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS

        /* Determine if notification is required.  */
        if (semaphore_put_notify != TX_NULL)
        {

            /* Yes, call the appropriate notify callback function.  */
            (semaphore_put_notify)(semaphore_ptr);
        }
#endif
    }

    /* Return successful completion.  */
    return(TX_SUCCESS);
}
```