(163条消息) ThreadX内核源码分析 - 动态内存管理_arm7star的博客-CSDN博客_threadx源码

blog.csdn.net/arm7star/article/details/123018855

ThreadX内核与μC / OS-II、NucleusPlus内核一样，都支持固定大小的内存块管理，也支持不固定大小的内存块管理。

固定大小的内存块管理是把一片内存分割成大小相同的多个内存块，以整块内存分配和释放，例如内存块大小为1024字节，那么每次只能申请释放1024字节的内存，一般只适合大小固定的内存块申请释放场景，类似linux的2的n次方的内存页申请，mmu只能管理页，不能把页分成更小的内存管理，因此要分配小的内存前，先按页分配，再在分配的页里面按byte分配内存。

1、block内存管理

1.1、block内存块介绍

ThreadX内核的内存块大致如下所示，将一块连续内存分割成n块连续大小相等的块，然后将这些块组成一个空闲链表，空闲内存块的前面部分信息就用于链表指针：

1.2、block内存申请_tx_block_allocate

申请内存主要是获取可用内存块，主要过程是：

- 有可用内存块，在可用内存块前面空间存储pool地址，返回存储pool地址后的下一个可用内存地址(并不把整个内存块返回给应用程序，释放内存块函数不带 pool参数，因此需要通过可用内存往前的一段固定偏移地址来获取pool信息；有的系统释放是明确指定内存块的pool，那么就可以不预留指向pool的内存)；
- 没有可用内存块，那么就可能要阻塞当前线程，阻塞时主要是将线程挂载到pool的挂起链表，然后将可用内存块地址(应用程序获取内存的指针及其他信息保存到线程控制块里面，释放内存的线程就可以直接修改等待内存线程的内存返回地址、返回状态，而不需要先把内存放回空闲链表，再唤醒等待线程，让唤醒线程重新去申请内存，这样效率比较低)。

_tx_block_allocate申请内存代码如下：

```
UINT  _tx_block_allocate(TX_BLOCK_POOL *pool_ptr, VOID **block_ptr, ULONG wait_option)
{

TX_INTERRUPT_SAVE_AREA

UINT                        status;
TX_THREAD                   *thread_ptr;
UCHAR                       *work_ptr;
UCHAR                       *temp_ptr;
UCHAR                       **next_block_ptr;
UCHAR                       **return_ptr;
UINT                        suspended_count;
TX_THREAD                   *next_thread;
TX_THREAD                   *previous_thread;
#ifdef TX_ENABLE_EVENT_TRACE
TX_TRACE_BUFFER_ENTRY       *entry_ptr;
ULONG                       time_stamp =  ((ULONG) 0);
#endif
#ifdef TX_ENABLE_EVENT_LOGGING
UCHAR                       *log_entry_ptr;
ULONG                       upper_tbu;
ULONG                       lower_tbu;
#endif


    /* Disable interrupts to get a block from the pool.  */
    TX_DISABLE

#ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO

    /* Increment the total allocations counter.  */
```

```
    _tx_block_pool_performance_allocate_count++;
    /* Increment the number of allocations on this pool.  */
    pool_ptr -> tx_block_pool_performance_allocate_count++;
#endif
#ifdef TX_ENABLE_EVENT_TRACE
    /* If trace is enabled, save the current event pointer.  */
    entry_ptr =  _tx_trace_buffer_current_ptr;
    /* If trace is enabled, insert this event into the trace buffer.  */
    TX_TRACE_IN_LINE_INSERT(TX_TRACE_BLOCK_ALLOCATE, pool_ptr, 0, wait_option, pool_ptr -> tx_block_pool_available, TX_TRACE_BLOCK_POOL_EVENTS)
    /* Save the time stamp for later comparison to verify that
       the event hasn't been overwritten by the time the allocate
       call succeeds.  */
    if (entry_ptr != TX_NULL)
    {
        time_stamp =  entry_ptr -> tx_trace_buffer_entry_time_stamp;
    }
#endif
#ifdef TX_ENABLE_EVENT_LOGGING
    log_entry_ptr =  *(UCHAR **) _tx_el_current_event;
    /* Log this kernel call.  */
    TX_EL_BLOCK_ALLOCATE_INSERT
    /* Store -1 in the third event slot.  */
```

```
    *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_3_OFFSET)) =  (ULONG) -1;

    /* Save the time stamp for later comparison to verify that
       the event hasn't been overwritten by the time the allocate
       call succeeds.  */
    lower_tbu =  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_LOWER_OFFSET));
    upper_tbu =  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_UPPER_OFFSET));
#endif

    /* Determine if there is an available block.  */
    if (pool_ptr -> tx_block_pool_available != ((UINT) 0)) // 可用空闲内存块数量tx_block_pool_available不为0，有可用空闲内存块，那么获取内存块即可
    {

        /* Yes, a block is available.  Decrement the available count.  */
        pool_ptr -> tx_block_pool_available--; // 可用空闲内存块数量减1

        /* Pickup the current block pointer.  */
        work_ptr =  pool_ptr -> tx_block_pool_available_list; // 获取第一可用空闲内存块(所有内存大小都一样，只需要获取可用空闲内存块链表表头的第一个内存块即可)

        /* Return the first available block to the caller.  */
        temp_ptr =  TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR *))); // temp_ptr等于work_ptr加上一个指针(内存块的前面一个指针大小用与指向内存块所在的pool，ThreadX的内存块释放函数没有指定pool，所以要在内存块的前面保存pool)
        return_ptr =  TX_INDIRECT_VOID_TO_UCHAR_POINTER_CONVERT(block_ptr); // block_ptr强制转换为UCHAR **类型(类似malloc，内存指针都是void *)
        *return_ptr =  temp_ptr; // 申请到的内存地址

        /* Modify the available list to point at the next block in the pool. */
        next_block_ptr =  TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr); // 下一个可用空闲内存块
        pool_ptr -> tx_block_pool_available_list =  *next_block_ptr; // pool的可用空闲内存块链表指向下一个空闲内存块

```

```
        /* Save the pool's address in the block for when it is released!  */
        temp_ptr =  TX_BLOCK_POOL_TO_UCHAR_POINTER_CONVERT(pool_ptr); // pool的地址
        *next_block_ptr =  temp_ptr; // pool地址写入申请的内存块的前面(释放内存块时，找到对应的pool，放回到对应的pool)
#ifdef TX_ENABLE_EVENT_TRACE
        /* Check that the event time stamp is unchanged.  A different
           timestamp means that a later event wrote over the byte
           allocate event.  In that case, do nothing here.  */
        if (entry_ptr != TX_NULL)
        {
            /* Is the time stamp the same?  */
            if (time_stamp == entry_ptr -> tx_trace_buffer_entry_time_stamp)
            {
                /* Timestamp is the same, update the entry with the address.  */
#ifdef TX_MISRA_ENABLE
                entry_ptr -> tx_trace_buffer_entry_info_2 = TX_POINTER_TO_ULONG_CONVERT(*block_ptr);
#else
                entry_ptr -> tx_trace_buffer_entry_information_field_2 = TX_POINTER_TO_ULONG_CONVERT(*block_ptr);
#endif
            }
        }
#endif
#ifdef TX_ENABLE_EVENT_LOGGING
        /* Store the address of the allocated block.  */
        *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_3_OFFSET)) =  (ULONG) *block_ptr;
```

```
#endif

        /* Set status to success.  */
        status =  TX_SUCCESS;

        /* Restore interrupts.  */
        TX_RESTORE // 申请到了内存，开启中断
    }
    else // 没有可用空闲内存块的清空下
    {

        /* Default the return pointer to NULL.  */
        return_ptr =   TX_INDIRECT_VOID_TO_UCHAR_POINTER_CONVERT(block_ptr);
        *return_ptr =  TX_NULL; // 设置返回内存的地址为null

        /* Determine if the request specifies suspension.  */
        if (wait_option != TX_NO_WAIT) // 非TX_NO_WAIT，也就是申请内存的线程在没申请到内存时要等待有可用内存或者等待超时
        {

            /* Determine if the preempt disable flag is non-zero.  */
            if (_tx_thread_preempt_disable != ((UINT) 0)) // 如果禁用了抢占，那么线程不能阻塞，否则其他线程也得不到调度，暂用内存块的线程得不到调度内存就不可能释放，内核会进入死循环
            {

                /* Suspension is not allowed if the preempt disable flag is non-zero at this point, return error completion.  */
                status =  TX_NO_MEMORY; // 申请不到内存又不能进入阻塞状态，设置没又内存可用，后面返回上一级函数

                /* Restore interrupts.  */
                TX_RESTORE
            }
```

```
            else // 允许抢占，线程可以进入阻塞状态
            {

                /* Prepare for suspension of this thread.  */

#ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO

                /* Increment the total suspensions counter.  */
                _tx_block_pool_performance_suspension_count++;

                /* Increment the number of suspensions on this pool.  */
                pool_ptr -> tx_block_pool_performance_suspension_count++;
#endif

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

                /* Setup cleanup routine pointer.  */
                thread_ptr -> tx_thread_suspend_cleanup =  &(_tx_block_pool_cleanup); // 设置挂起清理函数指针(阻塞被唤醒后执行的函数)

                /* Setup cleanup information, i.e. this pool control
                   block.  */
                thread_ptr -> tx_thread_suspend_control_block =  (VOID *) pool_ptr; // pool信息

                /* Save the return block pointer address as well.  */
                thread_ptr -> tx_thread_additional_suspend_info =  (VOID *) block_ptr; // 内存地址

#ifndef TX_NOT_INTERRUPTABLE

                /* Increment the suspension sequence number, which is used to identify
```

```
                this suspension event.  */
            thread_ptr -> tx_thread_suspension_sequence++;
#endif

            /* Pickup the number of suspended threads.  */
            suspended_count =  (pool_ptr -> tx_block_pool_suspended_count);

            /* Increment the number of suspended threads.  */
            (pool_ptr -> tx_block_pool_suspended_count)++; // 因等待内存而挂起的线程数加1

            /* Setup suspension list.  */
            if (suspended_count == TX_NO_SUSPENSIONS) // 如果当前线程是第一个等待内存的线程，把当前线程挂载到pool的挂起链表tx_block_pool_suspension_list即可
            {

                /* No other threads are suspended.  Setup the head pointer and
                   just setup this threads pointers to itself.  */
                pool_ptr -> tx_block_pool_suspension_list =     thread_ptr;
                thread_ptr -> tx_thread_suspended_next =        thread_ptr;
                thread_ptr -> tx_thread_suspended_previous =    thread_ptr;
            }
            else // 还有其他线程也在等待内存块，当前线程添加到tx_block_pool_suspension_list末尾即可
            {

                /* This list is not NULL, add current thread to the end. */
                next_thread =                                   pool_ptr -> tx_block_pool_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;
                }

                /* Set the state to suspended.  */
                thread_ptr -> tx_thread_state =       TX_BLOCK_MEMORY; // 设置线程阻塞状态(之前有介绍挂起线程操作，如果线程处于阻塞状态，那么需要延迟挂起，否则需要将线程从tx_block_pool_suspension_list删除，当有可用内存块的时候，可用内存就不会分配给不在等待链表里面的线程)

#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;

                /* Setup the timeout period.  */
                thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks =  wait_option; // 超时时间(_tx_thread_system_suspend会根据该值决定是否启动定时器，线程如果在超时时间内没有获取到内存，那么超时定时器将被调用，线程将被唤醒)

                /* Temporarily disable preemption.  */
                _tx_thread_preempt_disable++;

                /* Restore interrupts.  */
                TX_RESTORE

                /* Call actual thread suspension routine.  */
```

```
                _tx_thread_system_suspend(thread_ptr); // 挂起当前线程
#endif

#ifdef TX_ENABLE_EVENT_TRACE

                /* Check that the event time stamp is unchanged.  A different
                   timestamp means that a later event wrote over the byte
                   allocate event.  In that case, do nothing here.  */
                if (entry_ptr != TX_NULL)
                {

                    /* Is the time-stamp the same?  */
                    if (time_stamp == entry_ptr -> tx_trace_buffer_entry_time_stamp)
                    {

                        /* Timestamp is the same, update the entry with the address.  */
#ifdef TX_MISRA_ENABLE
                        entry_ptr -> tx_trace_buffer_entry_info_2 = TX_POINTER_TO_ULONG_CONVERT(*block_ptr);
#else
                        entry_ptr -> tx_trace_buffer_entry_information_field_2 = TX_POINTER_TO_ULONG_CONVERT(*block_ptr);
#endif
                    }
                }
#endif

#ifdef TX_ENABLE_EVENT_LOGGING
                /* Check that the event time stamp is unchanged and the call is about
                   to return success.  A different timestamp means that a later event
```

```
                   wrote over the block allocate event.  A return value other than
                   TX_SUCCESS indicates that no block was available. In those cases,
                   do nothing here.  */
                if (lower_tbu ==  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_LOWER_OFFSET)) &&
                    upper_tbu ==  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_UPPER_OFFSET)) &&
                    ((thread_ptr -> tx_thread_suspend_status) == TX_SUCCESS))
                {

                    /* Store the address of the allocated block.  */
                    *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_3_OFFSET)) = (ULONG) *block_ptr;
                }
#endif

                /* Return the completion status.  */
                status =  thread_ptr -> tx_thread_suspend_status; // 线程挂起状态(释放内存块的线程会把内存块直接给等待内存的线程，也就是tx_thread_additional_suspend_info(block_ptr))
            }
        }
        else
        {

            /* Immediate return, return error completion.  */
            status =  TX_NO_MEMORY;

            /* Restore interrupts.  */
            TX_RESTORE
        }
    }

```

```
    /* Return completion status.  */

    return(status);

}
```

1.3、申请内存块超时

线程申请内存块超时，主要是超时定时器处理，超时定时器处理线程_tx_timer_thread_entry调用线程超时函数_tx_thread_timeout处理线程超时事件，_tx_thread_timeout调用线程超时的处理函数tx_thread_suspend_cleanup，对应等待内存的线程回调函数就指向_tx_block_pool_cleanup。

等待内存超时的清理函数，主要从等待链表删除超时线程，设置线程挂起返回状态(没有内存)，唤醒阻塞的线程，_tx_block_pool_cleanup代码实现如下：

```
VOID  _tx_block_pool_cleanup(TX_THREAD *thread_ptr, ULONG suspension_sequence)
{
#ifndef TX_NOT_INTERRUPTABLE
TX_INTERRUPT_SAVE_AREA
#endif
TX_BLOCK_POOL       *pool_ptr;
UINT                suspended_count;
TX_THREAD           *next_thread;
TX_THREAD           *previous_thread;
#ifndef TX_NOT_INTERRUPTABLE
    /* Disable interrupts to remove the suspended thread from the block pool. */
    TX_DISABLE
    /* Determine if the cleanup is still required.  */
    if (thread_ptr -> tx_thread_suspend_cleanup == &(_tx_block_pool_cleanup)) //再次检查tx_thread_suspend_cleanup函数，有可能超时处理过程，释放内存的线程已经把内存分配给了超时的线程(超时和内存释放同时发生，保证不了谁先执行)
    {
        /* Check for valid suspension sequence.  */
        if (suspension_sequence == thread_ptr -> tx_thread_suspension_sequence)
        {
            /* Setup pointer to block pool control block.  */
            pool_ptr =  TX_VOID_TO_BLOCK_POOL_POINTER_CONVERT(thread_ptr -> tx_thread_suspend_control_block); // 获取等待的pool
```

```
            /* Check for a NULL byte pool pointer.  */
            if (pool_ptr != TX_NULL) // pool不为空(再次核对pool相关信息)
            {
                /* Check for valid pool ID.  */
                if (pool_ptr -> tx_block_pool_id == TX_BLOCK_POOL_ID) // pool类型为块类型
                {
                    /* Determine if there are any thread suspensions.  */
                    if (pool_ptr -> tx_block_pool_suspended_count != TX_NO_SUSPENSIONS) // 是否有等待内存的线程(如果没有的话，那么需要唤醒的线程应该就不再等待内存块了；ThreadX很多开关中断操作，有些关联操作不保证原子操作，所以有多重检查)
                    {
#else
                        /* Setup pointer to block pool control block.  */
                        pool_ptr = TX_VOID_TO_BLOCK_POOL_POINTER_CONVERT(thread_ptr -> tx_thread_suspend_control_block); // 获取等待的pool
#endif
                        /* Yes, we still have thread suspension!  */
                        /* Clear the suspension cleanup flag.  */
                        thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL; // 清空线程挂起清理函数指针
                        /* Decrement the suspended count.  */
                        pool_ptr -> tx_block_pool_suspended_count--; // 线程超时，不再等待内存块，pool等待挂起计数器减1
                        /* Pickup the suspended count.  */
                        suspended_count =  pool_ptr -> tx_block_pool_suspended_count;
```

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

/* See if this is the only suspended thread on the list.  */

if (suspended_count == TX_NO_SUSPENSIONS) // 当前线程是最后一个等待内存块的线程，等待链表清空即可
{

    /* Yes, the only suspended thread.  */

    /* Update the head pointer.  */
    pool_ptr -> tx_block_pool_suspension_list = TX_NULL;
}
else // 还有其他线程等待内存块，将当前线程从等待链表删除
{

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

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

    /* Determine if we need to update the head pointer.  */
    if (pool_ptr -> tx_block_pool_suspension_list == thread_ptr)
    {

        /* Update the list head pointer.  */
```

```
                    pool_ptr -> tx_block_pool_suspension_list = next_thread;
                }
            }

            /* Now we need to determine if this cleanup is from a terminate, timeout,
               or from a wait abort.  */
            if (thread_ptr -> tx_thread_state == TX_BLOCK_MEMORY) // 检查是等待超时调用清理函数还是函数线程终止调用清理函数，如果是线程终止，那么不需要做其他处理
            {

                /* Timeout condition and the thread still suspended on the block pool.
                   Setup return error status and resume the thread.  */

#ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO

                /* Increment the total timeouts counter.  */
                _tx_block_pool_performance_timeout_count++;

                /* Increment the number of timeouts on this block pool.  */
                pool_ptr -> tx_block_pool_performance_timeout_count++;
#endif

                /* Setup return status.  */
                thread_ptr -> tx_thread_suspend_status = TX_NO_MEMORY; // 等待内存超时，设置返回状态为TX_NO_MEMORY，没有等到内存

#ifdef TX_NOT_INTERRUPTABLE

                /* Resume the thread!  */
```

```
                        _tx_thread_system_ni_resume(thread_ptr);
#else
                        /* Temporarily disable preemption.  */
                        _tx_thread_preempt_disable++;

                        /* Restore interrupts.  */
                        TX_RESTORE

                        /* Resume the thread!  */
                        _tx_thread_system_resume(thread_ptr); // 唤醒阻塞的等待内存的线程

                        /* Disable interrupts.  */
                        TX_DISABLE
#endif
                    }
#ifndef TX_NOT_INTERRUPTABLE
                }
            }
        }
    }
}

    /* Restore interrupts.  */
    TX_RESTORE
#endif
}
```

1.4、block内存释放_tx_block_release

与申请内存超时一样，_tx_block_release释放内存也是检查等待内存的线程的各种参数状态，最后把线程控制块里面记录的返回内存的地址取出来，把当前释放的内存地址赋值给等待线程的内存返回地址即可，然后设置获取内存成功，唤醒等待内存块的线程(唤醒线程

函数会把超时定时器去激活)；如果没有线程等待内存，那么将释放的内存返回空闲链表，插入空闲链表表头(内存块大小一样，下次应该比较块申请到该内存块，对cache性能应该有改善，如果分配一个没有使用过的cache，那么cache需要重新从内存加载)。

_tx_block_release代码比较简单，实现如下：

```
UINT  _tx_block_release(VOID *block_ptr)
{

TX_INTERRUPT_SAVE_AREA

TX_BLOCK_POOL       *pool_ptr;
TX_THREAD           *thread_ptr;
UCHAR               *work_ptr;
UCHAR               **return_block_ptr;
UCHAR               **next_block_ptr;
UINT                suspended_count;
TX_THREAD           *next_thread;
TX_THREAD           *previous_thread;


    /* Disable interrupts to put this block back in the pool.  */
    TX_DISABLE

    /* Pickup the pool pointer which is just previous to the starting
       address of the block that the caller sees.  */
    work_ptr =        TX_VOID_TO_UCHAR_POINTER_CONVERT(block_ptr);
    work_ptr =        TX_UCHAR_POINTER_SUB(work_ptr, (sizeof(UCHAR *)));
    next_block_ptr =  TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr);
    pool_ptr =        TX_UCHAR_TO_BLOCK_POOL_POINTER_CONVERT((*next_block_ptr));

#ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO

    /* Increment the total releases counter.  */
    _tx_block_pool_performance_release_count++;

    /* Increment the number of releases on this pool.  */
```

```
    pool_ptr -> tx_block_pool_performance_release_count++;
#endif

    /* If trace is enabled, insert this event into the trace buffer.  */
    TX_TRACE_IN_LINE_INSERT(TX_TRACE_BLOCK_RELEASE, pool_ptr, TX_POINTER_TO_ULONG_CONVERT(block_ptr), pool_ptr -> tx_block_pool_suspended_count, TX_POINTER_TO_ULONG_CONVERT(&work_ptr), TX_TRACE_BLOCK_POOL_EVENTS)

    /* Log this kernel call.  */
    TX_EL_BLOCK_RELEASE_INSERT

    /* Determine if there are any threads suspended on the block pool.  */
    thread_ptr =  pool_ptr -> tx_block_pool_suspension_list;
    if (thread_ptr != TX_NULL)
    {

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

        /* Decrement the number of threads suspended.  */
        (pool_ptr -> tx_block_pool_suspended_count)--;

        /* Pickup the suspended count.  */
        suspended_count =  (pool_ptr -> tx_block_pool_suspended_count);

        /* See if this is the only suspended thread on the list.  */
        if (suspended_count == TX_NO_SUSPENSIONS)
        {

            /* Yes, the only suspended thread.  */

            /* Update the head pointer.  */
            pool_ptr -> tx_block_pool_suspension_list =  TX_NULL;
        }
```

```
        else
        {

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

            /* Update the list head pointer.  */
            next_thread =                                thread_ptr -> tx_thread_suspended_next;
            pool_ptr -> tx_block_pool_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;
        }

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

        /* Clear cleanup routine to avoid timeout.  */
        thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL;

        /* Return this block pointer to the suspended thread waiting for
           a block.  */
        return_block_ptr =  TX_VOID_TO_INDIRECT_UCHAR_POINTER_CONVERT(thread_ptr -> tx_thread_additional_suspend_info);
        work_ptr =          TX_VOID_TO_UCHAR_POINTER_CONVERT(block_ptr);
        *return_block_ptr =  work_ptr;

        /* Put return status into the thread control block.  */
        thread_ptr -> tx_thread_suspend_status =  TX_SUCCESS;

#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++;

        /* Restore interrupts.  */
        TX_RESTORE

        /* Resume thread.  */
        _tx_thread_system_resume(thread_ptr);
#endif
    }
    else
    {

        /* No thread is suspended for a memory block.  */

        /* Put the block back in the available list.  */
        *next_block_ptr =  pool_ptr -> tx_block_pool_available_list;

        /* Adjust the head pointer.  */
        pool_ptr -> tx_block_pool_available_list =  work_ptr;

        /* Increment the count of available blocks.  */
        pool_ptr -> tx_block_pool_available++;

```

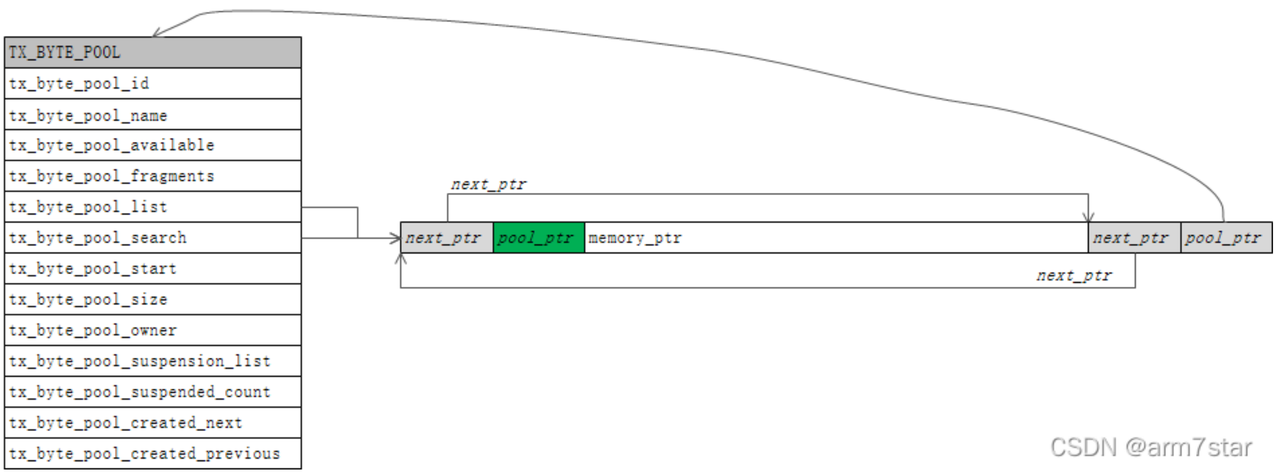
```
        /* Restore interrupts.  */
        TX_RESTORE
    }

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

2、byte内存管理

2.1、byte内存管理介绍

byte内存管理就是分配内存的大小不固定，内存块大小不固定，byte内存管理将内存初始化两个节点，最后一个节点为不可用内存，所有的内存块组成一个循环链表，相邻地址的内存块在链表中紧挨着；与block块内存管理不同的是，非空闲内存块也在链表中，只是多空闲与否的标记以及下一个内存块地址的指针，byte内存块申请过程会把一个大的空闲内存块一分为二，产生较多小的不连续的空闲内存块，为了避免太多碎片，内存块不够的时候需要合并相邻的空闲内存块，因此所有内存内存块是按地址链接在一个链表里面的；内存申请的时候只标记内存块非空闲，并不会改变链表结构。

初始化的内存链表大致如下(下面的pool_ptr在内存块空闲的时候，里面的值是空闲标记，内存被申请的时候，里面是指向pool的指针，空闲标记的值比较特殊，不会与内存地址冲突，因此可以有两种用处；释放内存的时候需要获取pool指针，这样才能更新pool的信息)：



2.2、byte内存申请_tx_byte_allocate

_tx_byte_allocate申请内存对内存链表遍历需要较长时间，中间会打开中断，避免阻塞中断；

对于 并发操作，pool的操作增加了一个tx_byte_pool_owner记录最近一次操作pool的线程，开中断期间：

- 如果有别的线程也申请或者释放内存，那么tx_byte_pool_owner会被改变，当前线程检查到tx_byte_pool_owner不指向自己的线程，那么就可以判断有其他线程操作了pool，pool可能发生了变化，还没查找完pool的话，就需要重新查找pool，没有找到可用内存块的话，也需要重新查找pool(线程还没真正睡眠，修改pool的线程并不知道有线程等待内存，如果线程进入睡眠，即使有可用空闲内存，要是没有其他线程释放内存的话，阻塞的线程是感知不到内存变化的，就不会被唤醒)；
- 如果没有其他线程操作pool，那么线程可以继续之前查找的结果接着查找，没有找到可用空闲内存块的话就阻塞自己，挂载到pool等待链表，有别的线程释放内存的话，就会检查是否有空闲内存满足等待线程的内存，如果有就会给阻塞线程分配内存并唤醒阻塞线程。

_tx_byte_allocate代码实现如下：

```
UINT  _tx_byte_allocate(TX_BYTE_POOL *pool_ptr, VOID **memory_ptr, ULONG memory_size,  ULONG wait_option)
{
TX_INTERRUPT_SAVE_AREA
UINT                        status;
TX_THREAD                   *thread_ptr;
UCHAR                       *work_ptr;
UINT                        suspended_count;
TX_THREAD                   *next_thread;
TX_THREAD                   *previous_thread;
UINT                        finished;
#ifdef TX_ENABLE_EVENT_TRACE
TX_TRACE_BUFFER_ENTRY       *entry_ptr;
ULONG                       time_stamp =  ((ULONG) 0);
#endif
#ifdef TX_ENABLE_EVENT_LOGGING
UCHAR                       *log_entry_ptr;
ULONG                       upper_tbu;
ULONG                       lower_tbu;
#endif
    /* Round the memory size up to the next size that is evenly divisible by
       an ALIGN_TYPE (this is typically a 32-bit ULONG).  This guarantees proper alignment.  */
    memory_size = (((memory_size + (sizeof(ALIGN_TYPE)))-((ALIGN_TYPE) 1))/(sizeof(ALIGN_TYPE))) * (sizeof(ALIGN_TYPE));
    /* Disable interrupts.  */
    TX_DISABLE
```

```
    /* Pickup thread pointer.  */
    TX_THREAD_GET_CURRENT(thread_ptr)

#ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

    /* Increment the total allocations counter.  */
    _tx_byte_pool_performance_allocate_count++;

    /* Increment the number of allocations on this pool.  */
    pool_ptr -> tx_byte_pool_performance_allocate_count++;
#endif

#ifdef TX_ENABLE_EVENT_TRACE

    /* If trace is enabled, save the current event pointer.  */
    entry_ptr =  _tx_trace_buffer_current_ptr;

    /* If trace is enabled, insert this event into the trace buffer.  */
    TX_TRACE_IN_LINE_INSERT(TX_TRACE_BYTE_ALLOCATE, pool_ptr, 0, memory_size, wait_option, TX_TRACE_BYTE_POOL_EVENTS)

    /* Save the time stamp for later comparison to verify that
       the event hasn't been overwritten by the time the allocate
       call succeeds.  */
    if (entry_ptr != TX_NULL)
    {

        time_stamp =  entry_ptr -> tx_trace_buffer_entry_time_stamp;
    }
#endif

#ifdef TX_ENABLE_EVENT_LOGGING
```

```
    log_entry_ptr =  *(UCHAR **) _tx_el_current_event;

    /* Log this kernel call.  */
    TX_EL_BYTE_ALLOCATE_INSERT

    /* Store -1 in the fourth event slot.  */
    *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_4_OFFSET)) =  (ULONG) -1;

    /* Save the time stamp for later comparison to verify that
       the event hasn't been overwritten by the time the allocate
       call succeeds.  */
    lower_tbu =  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_LOWER_OFFSET));
    upper_tbu =  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_UPPER_OFFSET));
#endif

    /* Set the search finished flag to false.  */
    finished =  TX_FALSE;

    /* Loop to handle cases where the owner of the pool changed.  */
    do // 与block内存分配不同的是，block内存大小固定，释放内存时，肯定可以把释放的内存给另外一个等待内存的线程，但是byte内存释放时，释放的大小并不一定满足等待内存线程需要的内存的大小，因此有更多内存的时候，需要等待内存的线程自己去检查是否有足够空闲内存，所以这里是不断检查是否有足够空闲内存
    {

        /* Indicate that this thread is the current owner.  */
        pool_ptr -> tx_byte_pool_owner =  thread_ptr; // 记录当前线程在操作该pool(主要是用于检查是否有别的线程操作了pool，如果没有找到合适大小的可用空闲内存块，在进入阻塞前，如果有其他线程操作了pool，那么可能pool里面可能存在可用空闲内存块，如果不再去检查的话也没有其他线程释放内存的话，当前线程可能就永远不会被唤醒了，只有内存释放的时候会去检查唤醒等待内存的线程)

        /* Restore interrupts.  */
        TX_RESTORE // 允许中断(byte内存申请使用频繁，耗时比较长，不能简单的互斥访问，否则并发度很低，影响性能)

```

```
        /* At this point, the executing thread owns the pool and can perform a search
           for free memory.  */
        work_ptr =  _tx_byte_pool_search(pool_ptr, memory_size); // 查找memory_size大小的空闲内存(_tx_byte_pool_search的时候会合并拆分空闲链表)

        /* Optional processing extension.  */
        TX_BYTE_ALLOCATE_EXTENSION

        /* Lockout interrupts.  */
        TX_DISABLE

        /* Determine if we are finished.  */
        if (work_ptr != TX_NULL) // 如果找到了可用空闲内存，那么查找结束，work_ptr就是申请到的内存的地址
        {

            /* Yes, we have found a block the search is finished.  */
            finished =  TX_TRUE;
        }
        else // 没有找到可用的内存
        {

            /* No block was found, does this thread still own the pool?  */
            if (pool_ptr -> tx_byte_pool_owner == thread_ptr) // 有别的线程修改了tx_byte_pool_owner，对pool进行了操作，如果别的线程释放了内存，那么要再次检查是否有可用内存(没有找到合适内存后会打开中断，如果开中断后有高优先级新城就绪或者其他情况导致有内存被释放，那么这里需要重新查找可用内存)，如果pool没有被修改，那么就要进入睡眠，此时中断已经关闭，不会有线程修改pool了
            {

                /* Yes, then we have looked through the entire pool and haven't found the memory.  */
                finished =  TX_TRUE; // 没有其他线程修改pool，不用再检查释放有可用内存
            }
```

```
        }
    } while (finished == TX_FALSE); // finished为TX_FALSE，表明没有获取到内存并且pool被修改了，那么再次检查是否有可用内存
    /* Copy the pointer into the return destination.  */
    *memory_ptr =  (VOID *) work_ptr; // 申请到的内存或者null
    /* Determine if memory was found.  */
    if (work_ptr != TX_NULL) // 申请到了内存，返回成功，if里面很多代码都是性能统计或者其他trace的代码，不用过多查看
    {
#ifdef TX_ENABLE_EVENT_TRACE
        /* Check that the event time stamp is unchanged.  A different
           timestamp means that a later event wrote over the byte
           allocate event.  In that case, do nothing here.  */
        if (entry_ptr != TX_NULL)
        {
            /* Is the timestamp the same?  */
            if (time_stamp == entry_ptr -> tx_trace_buffer_entry_time_stamp)
            {
                /* Timestamp is the same, update the entry with the address.  */
#ifdef TX_MISRA_ENABLE
                entry_ptr -> tx_trace_buffer_entry_info_2 = TX_POINTER_TO_ULONG_CONVERT(*memory_ptr);
#else
                entry_ptr -> tx_trace_buffer_entry_information_field_2 = TX_POINTER_TO_ULONG_CONVERT(*memory_ptr);
#endif
            }
```

```
        }
#endif

#ifdef TX_ENABLE_EVENT_LOGGING
        /* Check that the event time stamp is unchanged.  A different
           timestamp means that a later event wrote over the byte
           allocate event.  In that case, do nothing here.  */
        if (lower_tbu ==  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_LOWER_OFFSET)) &&
            upper_tbu ==  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_UPPER_OFFSET)))
        {
            /* Store the address of the allocated fragment.  */
            *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_4_OFFSET)) =  (ULONG) *memory_ptr;
        }
#endif

        /* Restore interrupts.  */
        TX_RESTORE

        /* Set the status to success.  */
        status =  TX_SUCCESS;
    }
    else // 没有找到可用内存，检查是否等待内存还是直接返回失败
    {

        /* No memory of sufficient size was found...  */

        /* Determine if the request specifies suspension.  */
        if (wait_option != TX_NO_WAIT) // 有等待选项
        {

```

```
            /* Determine if the preempt disable flag is non-zero.  */
            if (_tx_thread_preempt_disable != ((UINT) 0)) // 检查是否禁止抢占，如果禁止抢占的话，线程不能阻塞，否则会禁止其他线程的调度，其他就绪线程都没办法执行了
            {

                /* Suspension is not allowed if the preempt disable flag is non-zero at this point - return error completion.  */
                status =  TX_NO_MEMORY; // 禁止抢占，不能阻塞，返回申请内存失败即可

                /* Restore interrupts.  */
                TX_RESTORE
            }
            else // 阻塞等待内存，超时处理过程与block内存超时一样，别的线程释放内存时，如果有足够内存的话，也是由释放内存的线程把内存给等待内存的线程(合并/拆分空闲内存块等)
            {

                /* Prepare for suspension of this thread.  */

#ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

                /* Increment the total suspensions counter.  */
                _tx_byte_pool_performance_suspension_count++;

                /* Increment the number of suspensions on this pool.  */
                pool_ptr -> tx_byte_pool_performance_suspension_count++;
#endif

                /* Setup cleanup routine pointer.  */
                thread_ptr -> tx_thread_suspend_cleanup =  &(_tx_byte_pool_cleanup);

                /* Setup cleanup information, i.e. this pool control
                   block.  */
```

```
                thread_ptr -> tx_thread_suspend_control_block =  (VOID *) pool_ptr;

                /* Save the return memory pointer address as well.  */
                thread_ptr -> tx_thread_additional_suspend_info =  (VOID *) memory_ptr; // 别的线程释放内存时，如果有可用内存分配给阻塞的线程，那么会把申请到的内存赋值给memory_ptr(被释放的内存可能满足内存大小，或者被释放内存合并相邻空闲块后的空闲内存块可能满足内存大小，所以释放内存时检查是否有合适内存效率应该高一些)

                /* Save the byte size requested.  */
                thread_ptr -> tx_thread_suspend_info =  memory_size; // 需要申请的内存大小

#ifndef TX_NOT_INTERRUPTABLE

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

                /* Pickup the number of suspended threads.  */
                suspended_count =  pool_ptr -> tx_byte_pool_suspended_count; // 后续代码跟block阻塞挂起时原理一样

                /* Increment the suspension count.  */
                (pool_ptr -> tx_byte_pool_suspended_count)++;

                /* Setup suspension list.  */
                if (suspended_count == TX_NO_SUSPENSIONS)
                {

                    /* No other threads are suspended.  Setup the head pointer and
                       just setup this threads pointers to itself.  */
                    pool_ptr -> tx_byte_pool_suspension_list =      thread_ptr;
```

```
                    thread_ptr -> tx_thread_suspended_next =        thread_ptr;
                    thread_ptr -> tx_thread_suspended_previous =    thread_ptr;
                }
                else
                {

                    /* This list is not NULL, add current thread to the end. */
                    next_thread =                                   pool_ptr -> tx_byte_pool_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;
                }

                /* Set the state to suspended.  */
                thread_ptr -> tx_thread_state =       TX_BYTE_MEMORY;

#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;

```

```
                /* Setup the timeout period.  */
                thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks =  wait_option;

                /* Temporarily disable preemption.  */
                _tx_thread_preempt_disable++;

                /* Restore interrupts.  */
                TX_RESTORE

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

#ifdef TX_ENABLE_EVENT_TRACE

                /* Check that the event time stamp is unchanged.  A different
                   timestamp means that a later event wrote over the byte
                   allocate event.  In that case, do nothing here.  */
                if (entry_ptr != TX_NULL)
                {

                    /* Is the timestamp the same?  */
                    if (time_stamp == entry_ptr -> tx_trace_buffer_entry_time_stamp)
                    {

                        /* Timestamp is the same, update the entry with the address.  */
#ifdef TX_MISRA_ENABLE
                        entry_ptr -> tx_trace_buffer_entry_info_2 = TX_POINTER_TO_ULONG_CONVERT(*memory_ptr);
#else
```

```
                       entry_ptr -> tx_trace_buffer_entry_information_field_2 = TX_POINTER_TO_ULONG_CONVERT(*memory_ptr);
#endif
                    }
                }
#endif

#ifdef TX_ENABLE_EVENT_LOGGING
                /* Check that the event time stamp is unchanged.  A different
                   timestamp means that a later event wrote over the byte
                   allocate event.  In that case, do nothing here.  */
                if (lower_tbu ==  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_LOWER_OFFSET)) &&
                    upper_tbu ==  *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_UPPER_OFFSET)))
                {

                    /* Store the address of the allocated fragment.  */
                    *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_4_OFFSET)) = (ULONG) *memory_ptr;
                }
#endif

                /* Return the completion status.  */
                status =  thread_ptr -> tx_thread_suspend_status; // 设置返回状态
            }
        }
        else
        {

            /* Restore interrupts.  */
            TX_RESTORE

```

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

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

2.3、byte空闲内存块查找_tx_byte_pool_search

ThreadX内核释放byte内存的时候，正常情况下只是将内存块标记为空闲，并不会立即合并空闲内存块，合并空闲内存块是在查找空闲内存块的_tx_byte_pool_search里面合并的，_tx_byte_pool_search在查找空闲内存块的时候，如果空闲内存块大小不满足申请的内存大小，那么检查下一个内存块是否空闲，如果空闲的话就合并下一个空闲内存块，如果满足申请内存的大小，就不检查下一个内存块是否可以合并，否则会浪费时间，影响效率。

_tx_byte_pool_search实现比较简单，基本就是找第一个满足申请内存大小的内存块：

- 空闲块大小不够的话，检查是否可以合并下一个内存块(如果下一个空闲的话)，内存块链表在pool初始化的时候就是非空闲的，所以最后一个内存块不会也不可能和第一个内存块合并，这样就减少了判断最后一个内存块的操作，代码逻辑就认为前一个内存块和后一个内存地址都是连续的；
- 空闲内存块大小过大的话，拆分成两个内存块，第一个就是申请的内存，第二个就是新的空闲内存块。

_tx_byte_pool_search实现代码如下：

```
UCHAR  *_tx_byte_pool_search(TX_BYTE_POOL *pool_ptr, ULONG memory_size)
{

TX_INTERRUPT_SAVE_AREA

UCHAR           *current_ptr;
UCHAR           *next_ptr;
UCHAR           **this_block_link_ptr;
UCHAR           **next_block_link_ptr;
ULONG           available_bytes;
UINT            examine_blocks;
UINT            first_free_block_found =  TX_FALSE;
TX_THREAD       *thread_ptr;
ALIGN_TYPE      *free_ptr;
UCHAR           *work_ptr;



    /* Disable interrupts.  */
    TX_DISABLE

    /* First, determine if there are enough bytes in the pool.  */
    if (memory_size >= pool_ptr -> tx_byte_pool_available) // 申请的内存大于等于所有可以用的内存大小(还要预留部分内存控制块，所以内存不够，返回null)
    {

        /* Restore interrupts.  */
        TX_RESTORE

        /* Not enough memory, return a NULL pointer.  */
        current_ptr =  TX_NULL;
    }
```

```
    else // 总的可用内存满足申请的内存大小(可能存在满足大小的连续内存块，需要查找一下)
    {

        /* Pickup thread pointer.  */
        TX_THREAD_GET_CURRENT(thread_ptr)

        /* Setup ownership of the byte pool.  */
        pool_ptr -> tx_byte_pool_owner =  thread_ptr; // 记录正在操作pool的线程，后面会开中断，如果开中断后，没有线程对pool操作，也就是pool没有被改变，那么可以接着之前的查找继续，否则得重新查找(也就是在多线程同时申请内存的时候效率比较低，查找空闲内存也不能阻塞高优先级线程申请内存，如果有高优先级线程申请内存，那么高优先级线程应该先获取内存)

        /* Walk through the memory pool in search for a large enough block.  */
        current_ptr =      pool_ptr -> tx_byte_pool_search; // 从tx_byte_pool_search开始查找空闲内存
        examine_blocks =   pool_ptr -> tx_byte_pool_fragments + ((UINT) 1); // tx_byte_pool_fragments内存碎片数量，也就是有多少个内存块(加1???多检查一次也不影响，暂时不考虑什么情况存有多检查一次的必要)
        available_bytes =  ((ULONG) 0);
        do
        {


#ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

            /* Increment the total fragment search counter.  */
            _tx_byte_pool_performance_search_count++;

            /* Increment the number of fragments searched on this pool.  */
            pool_ptr -> tx_byte_pool_performance_search_count++;
#endif

            /* Check to see if this block is free.  */
```

```
            work_ptr =  TX_UCHAR_POINTER_ADD(current_ptr, (sizeof(UCHAR *))); // 内存块加上UCHAR *大小的地址(current_ptr最前面保存的是下一个内存块的地址)
            free_ptr =  TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(work_ptr); // work_ptr指针转换为ALIGN_TYPE类型指针
            if ((*free_ptr) == TX_BYTE_BLOCK_FREE) // 检查是否是空闲内存
            {

                /* Determine if this is the first free block.  */
                if (first_free_block_found == TX_FALSE) // 没有找到过空闲块时，first_free_block_found为TX_FALSE，如果找到了空闲块，first_free_block_found为TX_TRUE(内核要记录第一个找到的空闲块，避免每次都从非空闲块开始查找空闲内存)
                {

                    /* This is the first free block.  */
                    pool_ptr->tx_byte_pool_search =  current_ptr; // 记录第一次找到的空闲内存块

                    /* Set the flag to indicate we have found the first free
                       block.  */
                    first_free_block_found =  TX_TRUE; // 已经找到了第一个空闲内存块
                }

                /* Block is free, see if it is large enough.  */

                /* Pickup the next block's pointer.  */
                this_block_link_ptr =  TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(current_ptr); // 空闲内存块的最前面记录的是下一个相邻内存块的地址
                next_ptr =             *this_block_link_ptr; // 获取下一个内存块的地址

                /* Calculate the number of bytes available in this block.  */
                available_bytes =   TX_UCHAR_POINTER_DIF(next_ptr, current_ptr); // 下一个内存块的地址减去当前内存块的地址就是当前内存块的大小
                available_bytes =   available_bytes - ((sizeof(UCHAR *)) + (sizeof(ALIGN_TYPE))); // 当前内存块大小减去一个指针以及块空闲类型的标志就是当前内存块可以返回给应用程序的可用内存大小
```

```c

                /* If this is large enough, we are done because our first-fit algorithm
                   has been satisfied!  */
                if (available_bytes >= memory_size) // 可用内存大小大于等于申请的内存，那么从当前内存块申请内存给应用程序即可
                {
                    /* Get out of the search loop!  */
                    break;
                }
                else // 内存块可用内存大小不够，检查下一个相邻的内存块是否空闲，是否可以合并
                {

                    /* Clear the available bytes variable.  */
                    available_bytes =  ((ULONG) 0);

                    /* Not enough memory, check to see if the neighbor is
                       free and can be merged.  */
                    work_ptr =  TX_UCHAR_POINTER_ADD(next_ptr, (sizeof(UCHAR *))); // 获取下一个相邻的内存块
                    free_ptr = TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(work_ptr); // 内存块空闲类型
                    if ((*free_ptr) == TX_BYTE_BLOCK_FREE) // 内存块空闲
                    {

                        /* Yes, neighbor block can be merged!  This is quickly accomplished
                           by updating the current block with the next blocks pointer.  */
                        next_block_link_ptr = TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(next_ptr); // 获取再下下一个内存块地址(两个内存块地址相减才能知道内存块的大小，内存控制块没有记录内存大小，只记录相邻的下一个内存块的地址)
                        *this_block_link_ptr =  *next_block_link_ptr; // 当前内存块current_ptr的下一个内存块地址指向下一个内存块next_ptr的下一个内存块(next_ptr内存被合并了)

```

```
                    /* Reduce the fragment total.  We don't need to increase the bytes
                       available because all free headers are also included in the available
                       count.  */
                    pool_ptr -> tx_byte_pool_fragments--; // 当前内存块与相邻的下一个内存块合并，内存碎片数减1

#ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

                    /* Increment the total merge counter.  */
                    _tx_byte_pool_performance_merge_count++;

                    /* Increment the number of blocks merged on this pool.  */
                    pool_ptr -> tx_byte_pool_performance_merge_count++;
#endif

                    /* See if the search pointer is affected.  */
                    if (pool_ptr -> tx_byte_pool_search ==  next_ptr) // 如果tx_byte_pool_search指向被合并的下一个空闲内存块，那么更新tx_byte_pool_search指向合并后的空闲内存块(旧的内存块不存在了)
                    {

                        /* Yes, update the search pointer.   */
                        pool_ptr -> tx_byte_pool_search =  current_ptr;
                    }
                }
                else // 下一个内存块不空闲
                {

                    /* Neighbor is not free so we can skip over it!  */
                    next_block_link_ptr =  TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(next_ptr);
                    current_ptr =  *next_block_link_ptr; // current_ptr指向下下一个内存块，下一个非空闲内存块
```

```

                        /* Decrement the examined block count to account for this one.  */
                        if (examine_blocks != ((UINT) 0)) // examine_blocks不为0，要检查的内存块数量减1(next_ptr检查过了，current_ptr的examine_blocks减1在后面)
                        {

                            examine_blocks--;

#ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

                            /* Increment the total fragment search counter.  */
                            _tx_byte_pool_performance_search_count++;

                            /* Increment the number of fragments searched on this pool.  */
                            pool_ptr -> tx_byte_pool_performance_search_count++;
#endif
                        }
                    }
                }
            }
            else // 当前内存块不是空闲内存块，更新current_ptr指向下一个内存块，检查下一个内存块
            {

                /* Block is not free, move to next block.  */
                this_block_link_ptr =  TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(current_ptr);
                current_ptr =  *this_block_link_ptr;
            }

            /* Another block has been searched... decrement counter.  */
            if (examine_blocks != ((UINT) 0)) // current_ptr检查过了，examine_blocks减1
```

```
            {

                examine_blocks--;
            }

            /* Restore interrupts temporarily.  */
            TX_RESTORE // 允许中断，这个检查空闲内存块耗费了一定时间，不能阻塞中断

            /* Disable interrupts.  */
            TX_DISABLE // 再次关闭中断

            /* Determine if anything has changed in terms of pool ownership.  */
            if (pool_ptr -> tx_byte_pool_owner != thread_ptr) // 在检查下一个内存块前，检查一下pool的owner是不是当前线程，如果不是当前线程，那么pool的内存可能就被其他线程修改了，current_ptr可能被合并无效了，要检查的内存碎片数量有不一样了，所以又得重新查找空闲内存块
            {

                /* Pool changed ownership in the brief period interrupts were
                   enabled.  Reset the search.  */
                current_ptr =      pool_ptr -> tx_byte_pool_search; // 重新获取空闲内存查找的起始内存块(tx_byte_pool_search一般指向一个空闲内存块，内存申请的时候，如果该内存块被本次申请了，那么tx_byte_pool_search就指向下一个内存块(不一定是空闲的)；内存释放的时候，如果释放的内存块地址小于tx_byte_pool_search，那么更新tx_byte_pool_search指向刚释放的内存块地址，也就是ThreadX内核尽量从内存块链表前面的内存块分配内存，这样就会优先拆分前面的空闲内存块，后面的内存块可能就比较大，尽量避免过多非连续的小内存碎片的产生)
                examine_blocks =   pool_ptr -> tx_byte_pool_fragments + ((UINT) 1);

                /* Setup our ownership again.  */
                pool_ptr -> tx_byte_pool_owner =  thread_ptr;
            }
        } while(examine_blocks != ((UINT) 0)); // 如果所有内存块都检查完了还没找到可用空闲内存块，那么退出循环

        /* Determine if a block was found.  If so, determine if it needs to be
```

```
193. 276            split.  */
194. 277         if (available_bytes != ((ULONG) 0)) // available_bytes不为0，表示找到了一个满足申请大小的内存块
195. 278         {
196. 279
197. 280             /* Determine if we need to split this block.  */
198. 281             if ((available_bytes - memory_size) >= ((ULONG) TX_BYTE_BLOCK_MIN)) // 可用内存减去申请的内存还大于等于TX_BYTE_BLOCK_MIN，也就是将当前内存块拆分后，剩余的内存还够一个空闲内存块(内核不拆分成太小的内存块，太小的内存块分配管理起来效率太低，干脆就把多一点点的内存块给应用程序即可，应用程序也不会访问多给的内存)
199. 282             {
200. 283
201. 284                 /* Split the block.  */ // 拆分当前内存块
202. 285                 next_ptr =  TX_UCHAR_POINTER_ADD(current_ptr, (memory_size + ((sizeof(UCHAR *)) + (sizeof(ALIGN_TYPE))))); // "下一个内存块地址指针、内存块释放空闲字段 + 申请的内存大小"这么多作为一个新的内存块，里面的可用内存返回给应用程序；接下来的内存划分成一个新的空闲内存块，next_ptr即指向新的空闲内存块的地址
203. 286
204. 287                 /* Setup the new free block.  */
205. 288                 next_block_link_ptr = TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(next_ptr); // 指针转换，没有改变值
206. 289                 this_block_link_ptr = TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(current_ptr); // 指针转换，没有改变值(current_ptr拆分前的下一个内存块地址，也就是拆分后的next_ptr内存块的下一个内存块地址)
207. 290                 *next_block_link_ptr =  *this_block_link_ptr; // next_ptr的下一个内存块地址，指向拆分前的current_ptr的下一个内存块地址
208. 291                 work_ptr =              TX_UCHAR_POINTER_ADD(next_ptr, (sizeof(UCHAR *))); // next_ptr + sizeof(UCHAR *)指向标记内存块是否空闲的地址
209. 292                 free_ptr = TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(work_ptr);
210. 293                 *free_ptr =             TX_BYTE_BLOCK_FREE; // next_ptr设置为空闲
211. 294
212. 295                 /* Increase the total fragment counter.  */
213. 296                 pool_ptr -> tx_byte_pool_fragments++;
214. 297
215. 298                 /* Update the current pointer to point at the newly created block.  */
```

```
                *this_block_link_ptr =  next_ptr; // current_ptr的下一个内存块地址指向拆分出来的next_ptr内存块地址

                /* Set available equal to memory size for subsequent calculation.  */
                available_bytes =  memory_size; // available_bytes等于拆分后申请到的内存块的可用内存大小

#ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

                /* Increment the total split counter.  */
                _tx_byte_pool_performance_split_count++;

                /* Increment the number of blocks split on this pool.  */
                pool_ptr -> tx_byte_pool_performance_split_count++;
#endif
            }

            /* In any case, mark the current block as allocated.  */
            work_ptr =              TX_UCHAR_POINTER_ADD(current_ptr, (sizeof(UCHAR *)));
            this_block_link_ptr =  TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr);
            *this_block_link_ptr =  TX_BYTE_POOL_TO_UCHAR_POINTER_CONVERT(pool_ptr); // 该地址指向pool地址(与标记内存释放空闲的标志占用同一个内存，pool_ptr不会等于空闲内存块标记，所以可以存pool指针，否则可能造成混乱；复用内存，这样就节省了一个内存空间)

            /* Reduce the number of available bytes in the pool.  */
            pool_ptr -> tx_byte_pool_available =  (pool_ptr -> tx_byte_pool_available - available_bytes) - ((sizeof(UCHAR *)) + (sizeof(ALIGN_TYPE))); // pool的可用内存空间减去申请出去的内存块大小

            /* Determine if the search pointer needs to be updated. This is only done
              if the search pointer matches the block to be returned.  */
```

```
            if (current_ptr == pool_ptr -> tx_byte_pool_search) // 更新tx_byte_pool_search指向当前内存块的下一个内存块(从tx_byte_pool_search开始查找空闲内存)，tx_byte_pool_search指向的内存块已经被申请了，下次不要再从这里开始查找，免得浪费时间(更新的时候不保证下一个内存块可用，所以前面的first_free_block_found也就是在查找过程中更新tx_byte_pool_search指向第一个空闲内存块)
            {

                /* Yes, update the search pointer to the next block.  */
                this_block_link_ptr = TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(current_ptr);
                pool_ptr -> tx_byte_pool_search =  *this_block_link_ptr; // 更新tx_byte_pool_search，tx_byte_pool_search指向的内存块已经被申请了，直接指向下一个内存块即可，不管下一个内存块是否空闲，由下次申请内存的线程来更新tx_byte_pool_search
            }

            /* Restore interrupts.  */
            TX_RESTORE

            /* Adjust the pointer for the application.  */
            current_ptr =  TX_UCHAR_POINTER_ADD(current_ptr, (((sizeof(UCHAR *)) + (sizeof(ALIGN_TYPE))))); // 修正current_ptr，修正前current_ptr包含了pool指针、下一个内存块的地址，跳过这些才是返回给应用程序的可用内存地址，也就是c语言malloc返回的地址
        }
        else // 查找完了所以内存块也没找到合适的空闲内存，返回空指针给上一级函数
        {

            /* Restore interrupts.  */
            TX_RESTORE

            /* Set current pointer to NULL to indicate nothing was found.  */
            current_ptr =  TX_NULL;
        }
    }

    /* Return the search pointer.  */
    return(current_ptr);
```

```
268. 351 }
```

2.4、byte内存释放_tx_byte_release

释放内存过程比较简单，与申请内存有很多重复的地方：

- 检查释放内存是否是byte pool里面的内存，不是的话需要返回错误；
- 释放内存，将内存块标记为空闲，更新pool的可用内存数量；如果释放的内存地址小于下次查找空闲的内存地址tx_byte_pool_search，更新tx_byte_pool_search指向被释放的空闲内存，使tx_byte_pool_search尽量从低地址空闲内存块开始查找，优先拆分链表前面的空闲内存块；
- 如果有线程等待内存，给表头线程分配内存，分配不到内存就跳出循环并返回，分配失败就把内存重新变成空闲状态(等待内存的线程链表有更新，分配内存的线程可能不在等待内存了)，重新给等待内存的线程链表表头线程分配内存，如果分配成功，则将线程从等待链表删除并唤醒线程，重新给等待内存的线程链表表头线程分配内存，直到没有找到满足表头线程内存大小的空闲内存(不继续给后续线程分配内存，优先给先申请内存的线程分配内存，如果给后续线程分配了内存，空闲内存块变得越来越小越来越少，先申请内存的线程可能更申请不到内存，这样不太公平)

_tx_byte_release释放内存，给等待内存的线程分配内存的实现代码如下：

```
UINT  _tx_byte_release(VOID *memory_ptr)
{

TX_INTERRUPT_SAVE_AREA

UINT                status;
TX_BYTE_POOL        *pool_ptr;
TX_THREAD           *thread_ptr;
UCHAR               *work_ptr;
UCHAR               *temp_ptr;
UCHAR               *next_block_ptr;
TX_THREAD           *susp_thread_ptr;
UINT                suspended_count;
TX_THREAD           *next_thread;
TX_THREAD           *previous_thread;
ULONG               memory_size;
ALIGN_TYPE          *free_ptr;
TX_BYTE_POOL        **byte_pool_ptr;
UCHAR               **block_link_ptr;
UCHAR               **suspend_info_ptr;


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

    /* Set the pool pointer to NULL.  */
    pool_ptr =  TX_NULL;

    /* Lockout interrupts.  */
    TX_DISABLE

```

```
    /* Determine if the memory pointer is valid.  */
    work_ptr =  TX_VOID_TO_UCHAR_POINTER_CONVERT(memory_ptr);
    if (work_ptr != TX_NULL) // 检查释放的内存是否为空指针(if分支主要获取memory_ptr所在的pool)
    {

        /* Back off the memory pointer to pickup its header.  */
        work_ptr =  TX_UCHAR_POINTER_SUB(work_ptr, ((sizeof(UCHAR *)) + (sizeof(ALIGN_TYPE)))); // 获取内存块的起始地址

        /* There is a pointer, pickup the pool pointer address.  */
        temp_ptr =  TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR *))); // 内存块起始地址加上下一个内存地址大小就是pool指针/free标志所在内存
        free_ptr =  TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(temp_ptr);
        if ((*free_ptr) != TX_BYTE_BLOCK_FREE) // 检查内存是否是空闲内存，如果重复释放内存，返回TX_PTR_ERROR
        {

            /* Pickup the pool pointer.  */
            temp_ptr =  TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR *))); // 获取pool指针所在内存地址，内存块结构为"|下一个内存块地址|pool指针/free标志|memory_ptr|"
            byte_pool_ptr =  TX_UCHAR_TO_INDIRECT_BYTE_POOL_POINTER(temp_ptr); // 指针转换
            pool_ptr =  *byte_pool_ptr; // 获取pool指针

            /* See if we have a valid pool pointer.  */
            if (pool_ptr == TX_NULL) // pool为空，那么memory_ptr不是pool里面的内存块，使用了错误的释放函数
            {

                /* Return pointer error.  */
                status =  TX_PTR_ERROR; // 返回错误
            }
            else // pool指针不为空，需要再次检查是否是真正的pool
            {
```

```
                /* See if we have a valid pool.  */
                if (pool_ptr -> tx_byte_pool_id != TX_BYTE_POOL_ID) // pool id对不上，pool指针不是真正指向有效的pool，返回错误
                {

                    /* Return pointer error.  */
                    status =  TX_PTR_ERROR;

                    /* Reset the pool pointer is NULL.  */
                    pool_ptr =  TX_NULL;
                }
            }
        }
        else
        {

            /* Return pointer error.  */
            status =  TX_PTR_ERROR;
        }
    }
    else
    {

        /* Return pointer error.  */
        status =  TX_PTR_ERROR;
    }

    /* Determine if the pointer is valid.  */
    if (pool_ptr == TX_NULL) // 没有找到memory_ptr所在的有效pool，memory_ptr不是从pool申请的，返回错误
    {

```

```
        /* Restore interrupts.  */
        TX_RESTORE
    }
    else // 释放memory_ptr
    {

        /* At this point, we know that the pointer is valid.  */

        /* Pickup thread pointer.  */
        TX_THREAD_GET_CURRENT(thread_ptr)

        /* Indicate that this thread is the current owner.  */
        pool_ptr -> tx_byte_pool_owner =  thread_ptr; // 标记当前线程正在操作pool(与申请内存作用一样)

#ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

        /* Increment the total release counter.  */
        _tx_byte_pool_performance_release_count++;

        /* Increment the number of releases on this pool.  */
        pool_ptr -> tx_byte_pool_performance_release_count++;
#endif

        /* If trace is enabled, insert this event into the trace buffer.  */
        TX_TRACE_IN_LINE_INSERT(TX_TRACE_BYTE_RELEASE, pool_ptr, TX_POINTER_TO_ULONG_CONVERT(memory_ptr), pool_ptr -> tx_byte_pool_suspended_count, pool_ptr -> tx_byte_pool_available, TX_TRACE_BYTE_POOL_EVENTS)

        /* Log this kernel call.  */
        TX_EL_BYTE_RELEASE_INSERT

        /* Release the memory.  */
```

```
        temp_ptr =   TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR *)));
        free_ptr =   TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(temp_ptr);
        *free_ptr =  TX_BYTE_BLOCK_FREE; // 将内存块标记为空闲

        /* Update the number of available bytes in the pool.  */
        block_link_ptr =  TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr);
        next_block_ptr =  *block_link_ptr;
        pool_ptr -> tx_byte_pool_available =
            pool_ptr -> tx_byte_pool_available + TX_UCHAR_POINTER_DIF(next_block_ptr, work_ptr); // "TX_UCHAR_POINTER_DIF(next_block_ptr, work_ptr)"当前内存块与下一个内存块之间的距离(包括下一个内存块地址指针等字段)作为当前空闲内存块的大小，当前释放的空闲内存块可能与相邻的空闲内存块合并，下一个内存地址等字段可能变成有效内存

        /* Determine if the free block is prior to current search pointer.  */
        if (work_ptr < (pool_ptr -> tx_byte_pool_search)) // 释放的空闲内存块地址小于下一次查找空闲内存块的起始地址，更新tx_byte_pool_search指向当前释放的空闲内存块；ThreadX尽量从低地址开始分配内存，这样位置越后的空闲内存块被拆分的机会就更少，也就是内存块就越大，有大内存申请的时候才可能获取到大内存；tx_byte_pool_search也不指向内存块表头，这样影响查找效率，所以内核是尽可能从前面的空闲内存块开始查找
        {

            /* Yes, update the search pointer to the released block.  */
            pool_ptr -> tx_byte_pool_search =  work_ptr; // 更新tx_byte_pool_search
        }

        /* Determine if there are threads suspended on this byte pool.  */
        if (pool_ptr -> tx_byte_pool_suspended_count != TX_NO_SUSPENSIONS) // 有线程在等待内存
        {

            /* Now examine the suspension list to find threads waiting for
               memory.  Maybe it is now available!  */
            while (pool_ptr -> tx_byte_pool_suspended_count != TX_NO_SUSPENSIONS) // 循环给表头线程分配内存(表头线程分配到内存后，下一个等待内存的线程将成为新的表头，循环过程就是一次给等待线程分配内存，如果分配不到就退出循环，不给后续等待线程分配内存)
```

```
            {
                /* Pickup the first suspended thread pointer.  */
                susp_thread_ptr =  pool_ptr -> tx_byte_pool_suspension_list; // 等待内存的线程链表
                /* Pickup the size of the memory the thread is requesting.  */
                memory_size =  susp_thread_ptr -> tx_thread_suspend_info; // 等待内存的大小(也就是malloc的内存大小)
                /* Restore interrupts.  */
                TX_RESTORE // 恢复中断，已经获取到了一个等待内存线程的数据，先试图给这个线程分配内存，别的线程还可以继续申请内存(tx_byte_pool_suspension_list是所有申请内存的线程共同访问的，关闭中断将导致其他线程不能申请内存)
                /* See if the request can be satisfied.  */
                work_ptr =  _tx_byte_pool_search(pool_ptr, memory_size); // 与申请内存时一样，查找memory_size的空闲内存块
                /* Optional processing extension.  */
                TX_BYTE_RELEASE_EXTENSION
                /* Disable interrupts.  */
                TX_DISABLE
                /* Indicate that this thread is the current owner.  */
                pool_ptr -> tx_byte_pool_owner =  thread_ptr;
                /* If there is not enough memory, break this loop!  */
                if (work_ptr == TX_NULL) // 没有找到可用内存块(这里不用判断是否有其他线程也操作了内存，如果别的线程释放了内存，那么也会试图给阻塞线程分配内存)
                {
                  /* Break out of the loop.  */
```

```
                break; // 退出循环，不继续分配内存(从这里看，ThreadX内核分配内存是优先给等待链表前面的线程分配内存，如果前面的线程都分配不到内存，那么就不给后面等待内存线程分配内存，即使内存够大)
            }
            /* Check to make sure the thread is still suspended.  */
            if (susp_thread_ptr ==  pool_ptr -> tx_byte_pool_suspension_list) // 再次检查susp_thread_ptr是否在阻塞链表里面(新的等待内存的线程会挂到等待链表末尾，不会改变表头，只有等待内存的线程超时，被从等待链表删除时，表头才可能变成其他线程)
            {
                /* Also, makes sure the memory size is the same.  */
                if (susp_thread_ptr -> tx_thread_suspend_info == memory_size) // 再次检查内存大小是否是刚才查找内存块的大小(存在等待内存的线程超时后再次加入等待链表的情况，也就是查找过程所有等待内存的线程都超时或者其他情况导致等待内存的线程链表都清空了，当前线程还没检测到，然后之前等待内存链表表头的线程又再次申请内存并且阻塞了，那么该线程又回到了阻塞链表的表头)
                {
                    /* Remove the suspended thread from the list.  */
                    /* Decrement the number of threads suspended.  */
                    pool_ptr -> tx_byte_pool_suspended_count--; // 等待内存的线程数量减1
                    /* Pickup the suspended count.  */
                    suspended_count =  pool_ptr -> tx_byte_pool_suspended_count;
                    /* See if this is the only suspended thread on the list.  */
                    if (suspended_count == TX_NO_SUSPENSIONS) // 如果没有其他线程等待内存，清空等待链表
                    {
                        /* Yes, the only suspended thread.  */
```

```
            /* Update the head pointer.  */
            pool_ptr -> tx_byte_pool_suspension_list =  TX_NULL; // 清空等待链表
        }
        else // 否则从等待链表删除当前分配内存的线程(表头线程)
        {

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

            /* Update the list head pointer.  */
            next_thread = susp_thread_ptr -> tx_thread_suspended_next;
            pool_ptr -> tx_byte_pool_suspension_list = next_thread;

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

        /* Prepare for resumption of the thread.  */

        /* Clear cleanup routine to avoid timeout.  */
        susp_thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL; // 清空tx_thread_suspend_cleanup，线程已经获取到内存了(这里还没去激活超时定时器，清空tx_thread_suspend_cleanup，即使定时器超时也不会有回调函数调用)

        /* Return this block pointer to the suspended thread waiting for
           a block.  */
        suspend_info_ptr = TX_VOID_TO_INDIRECT_UCHAR_POINTER_CONVERT(susp_thread_ptr -> tx_thread_additional_suspend_info); // 申请内存时的memory_ptr指针
```

```
                        *suspend_info_ptr =  work_ptr; // 设置memory_ptr，将work_ptr内存分配给线程

                        /* Clear the memory pointer to indicate that it was given to the suspended thread.  */
                        work_ptr =  TX_NULL; // 清空work_ptr(指示该内存已经被分配了，后面会用到work_ptr，如果等待内存链表表头变了或者内存大小不一致，那么需要将该内存块重新设置为空闲内存)

                        /* Put return status into the thread control block.  */
                        susp_thread_ptr -> tx_thread_suspend_status = TX_SUCCESS; // 线程返回状态(阻塞线程被唤醒后，使用这个作为获取内存的结果返回给上一级函数)

#ifdef TX_NOT_INTERRUPTABLE

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

                        /* Restore interrupts.  */
                        TX_RESTORE
#else
                        /* Temporarily disable preemption.  */
                        _tx_thread_preempt_disable++;

                        /* Restore interrupts.  */
                        TX_RESTORE

                        /* Resume thread.  */
                        _tx_thread_system_resume(susp_thread_ptr); // 唤醒已经分配到内存的线程
#endif

                        /* Lockout interrupts.  */
                        TX_DISABLE
```

```
                    }
                }

                /* Determine if the memory was given to the suspended thread.  */
                if (work_ptr != TX_NULL) // 等待内存的表头线程被改变了(内存没有分配成功，需要将内存重新变成空闲状态)
                {

                    /* No, it wasn't given to the suspended thread.  */

                    /* Put the memory back on the available list since this thread is no longer
                       suspended.  */
                    work_ptr =  TX_UCHAR_POINTER_SUB(work_ptr, (((sizeof(UCHAR *)) + (sizeof(ALIGN_TYPE)))));
                    temp_ptr =  TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR *)));
                    free_ptr =  TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(temp_ptr);
                    *free_ptr =  TX_BYTE_BLOCK_FREE;

                    /* Update the number of available bytes in the pool.  */
                    block_link_ptr =  TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr);
                    next_block_ptr =  *block_link_ptr;
                    pool_ptr -> tx_byte_pool_available =
                        pool_ptr -> tx_byte_pool_available + TX_UCHAR_POINTER_DIF(next_block_ptr, work_ptr);

                    /* Determine if the current pointer is before the search pointer.  */
                    if (work_ptr < (pool_ptr -> tx_byte_pool_search))
                    {

                        /* Yes, update the search pointer.  */
```

```
                        pool_ptr -> tx_byte_pool_search =  work_ptr; // work_ptr重新释放，更新tx_byte_pool_search
                    }
                }
            }

            /* Restore interrupts.  */
            TX_RESTORE

            /* Check for preemption.  */
            _tx_thread_system_preempt_check();
        }
        else // 没有线程等待内存，释放内存块标记为空闲即可，返回上一级函数
        {

            /* No, threads suspended, restore interrupts.  */
            TX_RESTORE
        }
    }

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