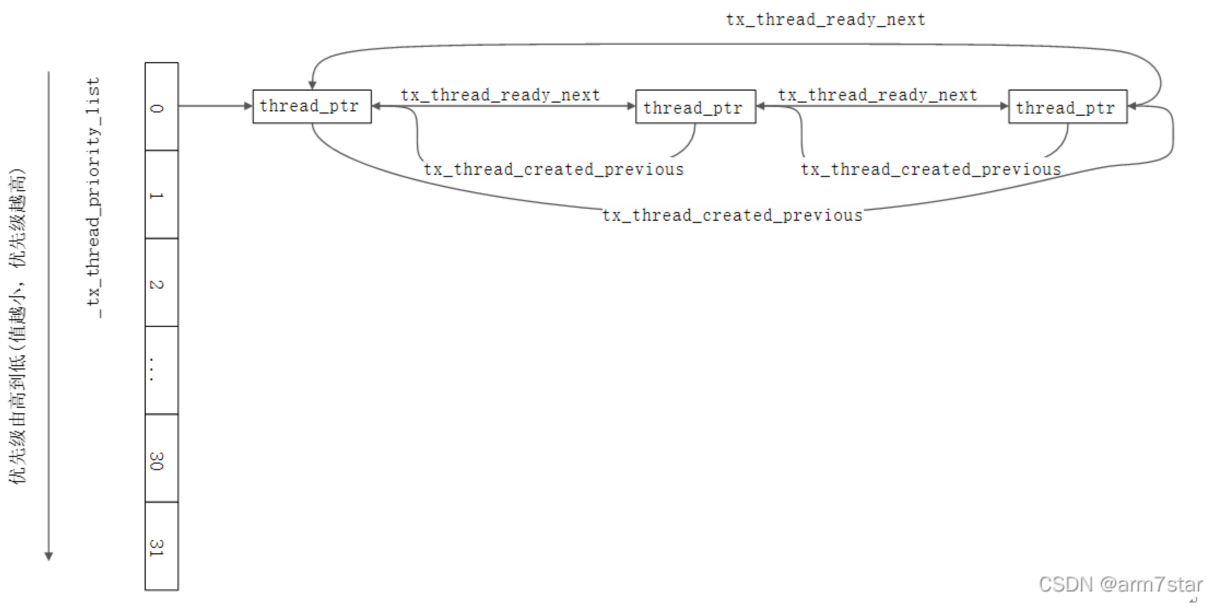
(163条消息) ThreadX内核源码分析 - 优先级及抢占阈值抢占(arm)_arm7star的博客-CSDN博客_threadx 优先级位图

1、就绪线程链表_tx_thread_priority_list

优先级越高值越低，同优先级线程按先进先出加入就绪线程链表，正在执行的线程始终在就绪线程链表的表头(线程调度时不会从就绪线程链表删除)。



2、抢占阈值threshold

线程的抢占阈值大于线程的优先级时，表示线程启用了抢占，也就是该线程可以抢占同优先级线程以及优先级比该线程抢占阈值低的线程；内核保证抢占阈值不会小于线程优先级，否则抢占阈值是错误的，因此只要抢占阈值不等与线程优先级就表示抢占阈值高于线程优先级；

没有启用抢占的线程按优先级及时间片调度，抢占阈值只对启用了抢占的线程正在执行时以及恢复执行时起作用：

- 启用抢占的线程正在执行时，如果时间片用尽或者有高优先级线程就绪，那么下一个可能需要调度的线程的优先级必须高于当前正在执行的线程的抢占阈值，被抢占时需要在_tx_thread_preempted_maps标记被抢占的线程(对应优先级位设置为1即可)，否则正在执行的线程抢占高优先级线程及同优先级线程继续执行；这里也就是抢占阈值对正在执行的线程有作用，不会判断被唤醒线程的抢占阈值是否高于正在执行线程的优先级及抢占阈值；

- 高优先级线程退出，高优先级就绪线程链表变为空(如果同优先级线程有就绪线程，那么同优先级就绪线程的优先级仍然高于被抢占的启用了抢占阈值线程的抢占阈值)，需要调度下一个高优先级就绪线程链表线程时，下一个高优先级的优先级可能高于当前退出线程的优先级也可能低于当前退出线程的优先级(正在执行线程启用了抢占，有高优先级线程在等待执行，或者没有高优先级线程就绪)，不管哪种情况，被抢占的启用了抢占的线程的抢占阈值都可能高于下一个需要调度的线程的优先级，如果有被抢占的启用了抢占的线程的抢占阈值高于下一个高优先级就绪线程，那么优先恢复启用了抢占被抢占出去的线程；这也就是为什么启用了抢占的线程被抢占的时候要在_tx_thread_preempted_maps里面标记了，也就是抢占阈值在被抢占线程在恢复执行时起作用的原因。

也可以以另外一种方式理解，可能更容易理解，线程被执行时，该线程的优先级就等于线程的抢占阈值threshold，该线程并没有加到threshold对应就绪线程链表，该线程被抢占出去的时候，下次并不能在threshold对应优先级链表找到，因此需要通过_tx_thread_preempted_maps找到threshold对应优先级线程，可以理解_tx_thread_preempted_maps就是一个临时的就绪线程链表，内核有两个就绪线程链表，调度线程时需要检查两个就绪线程链表而已。

3、抢占位图_tx_thread_preempted_maps

启用抢占的线程一定意义上是以threshold优先级在执行，被抢占时，threshold对应优先级链表找不到被抢占的线程，借助_tx_thread_preempted_maps来找到threshold优先级的线程；

启用抢占的正在执行的线程被抢占时，_tx_thread_preempted_maps对应优先级位设置为1，通过优先级找到线程的控制块，通过控制块找到线程的threshold，threshold也就是该线程执行时的动态优先级，有高优先级退出时，要检查threshold是否可以抢占其他就绪线程。

需要特别理解的是，_tx_thread_preempted_maps高优先级线程的优先级一定高于低优先级抢占阈值，把_tx_thread_preempted_maps当作就绪线程链表，那么可以理解_tx_thread_preempted_maps是按优先级排序的；启用抢占的高优先级线程被抢占的前提是高优先级线程需要执行，而执行的前提是，高优先级线程的优先级不能被_tx_thread_preempted_maps里面的线程的抢占阈值抢占，也就是高优先级线程的优先级高于_tx_thread_preempted_maps里面线程的抢占阈值，自然而然_tx_thread_preempted_maps高优先级线程的抢占阈值也高于_tx_thread_preempted_maps低优先级线程的抢占阈值。

4、优先级抢占

之前介绍时间片的时候介绍过启用抢占的线程时间片用尽时不会调度同优先级的下一个就绪线程，在此略过；内核只有在线程唤醒时，才可能导致当前执行线程的优先级不是最高的情况，因此，抢占只可能发生在线程唤醒时。

主要实现代码在_tx_thread_system_resume函数里面，_tx_thread_system_resume主要过程是：

- 去激活线程的定时器(这些定时器主要也是用来唤醒线程的，线程已经唤醒就不需要这些定时器了)

- 线程状态检查，多线程对同一个线程进行唤醒操作或者线程本来就不是挂起状态，对线程进行唤醒并不能真正唤醒线程
- 对唤醒线程检查是否可以抢占正在执行的线程，可以抢占的话，如果被抢占的线程已经启用了抢占，那么还得标记一下被抢占的线程，设置下一个执行的线程，返回系统，系统调度下一个需要执行的线程，否则加到就绪线程链表即可

tx_thread_system_resume代码实现如下：

```
VOID  _tx_thread_system_resume(TX_THREAD *thread_ptr)
#ifndef TX_NOT_INTERRUPTABLE
{

TX_INTERRUPT_SAVE_AREA

UINT            priority;
ULONG           priority_bit;
TX_THREAD       *head_ptr;
TX_THREAD       *tail_ptr;
TX_THREAD       *execute_ptr;
TX_THREAD       *current_thread;
ULONG           combined_flags;

#ifdef TX_ENABLE_EVENT_TRACE
TX_TRACE_BUFFER_ENTRY       *entry_ptr;
ULONG                       time_stamp =  ((ULONG) 0);
#endif

#if TX_MAX_PRIORITIES > 32
UINT            map_index;
#endif


#ifdef TX_ENABLE_STACK_CHECKING

    /* Check this thread's stack.  */
    TX_THREAD_STACK_CHECK(thread_ptr)
#endif

    /* Lockout interrupts while the thread is being resumed.  */
```

```
    TX_DISABLE
#ifndef TX_NO_TIMER
    /* Deactivate the timeout timer if necessary.  */
    if (thread_ptr -> tx_thread_timer.tx_timer_internal_list_head != TX_NULL) // tx_timer_internal_list_head指向超时链表表头，激活的定时器应该都会挂载到_tx_timer_list对应的超时定时器链表里面，如果tx_timer_internal_list_head为空，则说明定时器不在链表里面，也就是没有激活，否则定时器已经激活；ThreadX内核主要用这个定时器来唤醒线程，sleep/timeout等作用，既然这里唤醒线程，那么对应的定时器就不需要了，if分支就是去激活定时器，从超时定时器链表删除
    {
        /* Deactivate the thread's timeout timer.  */
        _tx_timer_system_deactivate(&(thread_ptr -> tx_thread_timer)); // 去激活线程超时定时器(与else分支比较，_tx_timer_system_deactivate没有清除tx_timer_internal_remaining_ticks；Nucleus Plus内核在发送信号给阻塞线程的时候，并不会停止线程的定时器，信号处理比较紧急，Nucleus Plus内核会暂时唤醒线程来处理信号，等信号处理完成后，线程仍然需要回到阻塞状态，等待定时器超时唤醒线程，因此Nucleus Plus内核的阻塞超时定时器并不会因为信号唤醒线程而停止)
    }
    else
    {
        /* Clear the remaining time to ensure timer doesn't get activated.  */
        thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks = ((ULONG) 0); // tx_timer_internal_remaining_ticks设置为0，确保不会获取到定时器激活状态(没有超时时间挂起线程的定时器的tx_timer_internal_remaining_ticks为TX_WAIT_FOREVER，该线程是不会激活定时器的，不会挂载到超时定时器链表)
    }
#endif
#ifdef TX_ENABLE_EVENT_TRACE
    /* If trace is enabled, save the current event pointer.  */
    entry_ptr =  _tx_trace_buffer_current_ptr;
#endif
```

```
    /* Log the thread status change.  */
    TX_TRACE_IN_LINE_INSERT(TX_TRACE_THREAD_RESUME, thread_ptr, thread_ptr -> tx_thread_state, TX_POINTER_TO_ULONG_CONVERT(&execute_ptr), TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr), TX_TRACE_INTERNAL_EVENTS)

#ifdef TX_ENABLE_EVENT_TRACE

    /* Save the time stamp for later comparison to verify that
       the event hasn't been overwritten by the time we have
       computed the next thread to execute.  */
    if (entry_ptr != TX_NULL)
    {

        /* Save time stamp.  */
        time_stamp =  entry_ptr -> tx_trace_buffer_entry_time_stamp;
    }
#endif

    /* Decrease the preempt disabled count.  */
    _tx_thread_preempt_disable--; //  禁止抢占计数器_tx_thread_preempt_disable减1(_tx_thread_system_resume进入前，已经禁止抢占了，但是中断还是开着的，如果不禁止抢占，那么唤醒过程就可能被抢占，唤醒过程就不能及时处理；这里虽然允许抢占了，但是中断还是关着的)

    /* Determine if the thread is in the process of suspending.  If so, the thread
       control block is already on the linked list so nothing needs to be done.  */
    if (thread_ptr -> tx_thread_suspending == TX_FALSE) // 唤醒非挂起过程中的线程(挂起中的线程被没有被真正挂起，还在就绪线程链表里面；真正被挂起的线程已经不在就绪线程链表里面，需要重新加入就绪线程链表)
    {

        /* Thread is not in the process of suspending. Now check to make sure the thread
           has not already been resumed.  */
```

```
        if (thread_ptr -> tx_thread_state != TX_READY) // 检查线程是否已经被唤醒(一般调用唤醒函数前都会对禁止抢占计数器加1，然后打开中断，虽然唤醒期间不会被抢占，但是不能保证中断服务程序不会唤醒线程，例如驱动中断服务程序的信号量就会唤醒等待信号量的线程)
        {

            /* No, now check to see if the delayed suspension flag is set.  */
            if (thread_ptr -> tx_thread_delayed_suspend == TX_FALSE) // 非延迟挂起状态(挂起一个非就绪/非挂起的线程，会设置tx_thread_delayed_suspend，tx_thread_delayed_suspend也就是有个挂起操作由于线程处于等待信号量或者其他状态而没能立即被挂起，需要等线程唤醒后再挂起，也就是当前函数_tx_thread_system_resume后面会使线程进入挂起状态)
            {

                /* Resume the thread!  */

                /* Make this thread ready.  */

                /* Change the state to ready.  */
                thread_ptr -> tx_thread_state =  TX_READY; // 线程状态变为就绪状态

                /* Pickup priority of thread.  */
                priority =  thread_ptr -> tx_thread_priority;

                /* Thread state change.  */
                TX_THREAD_STATE_CHANGE(thread_ptr, TX_READY)

                /* Log the thread status change.  */
                TX_EL_THREAD_STATUS_CHANGE_INSERT(thread_ptr, TX_READY)

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

                /* Increment the total number of thread resumptions.  */
                _tx_thread_performance_resume_count++;

```

```
                /* Increment this thread's resume count.  */
                thread_ptr -> tx_thread_performance_resume_count++;
#endif

                /* Determine if there are other threads at this priority that are
                   ready.  */
                head_ptr =  _tx_thread_priority_list[priority]; // 线程所在就绪线程链表
                if (head_ptr == TX_NULL) // 被唤醒线程的就绪线程链表没有其他就绪线程，那么就要一个先优先级的线程就绪，可能存在墙正在；如果过该优先级有其他其他就绪线程，被唤醒线程加到就绪线程链表末尾即可(内核调度的要么是当前优先级的第一个就绪线程或者更高优先级线程，当前被唤醒的线程怎么也不可能抢占线程被唤醒前的就绪线程)
                {

                    /* First thread at this priority ready.  Add to the front of the list.  */
                    _tx_thread_priority_list[priority] =       thread_ptr; // 该优先级的就绪线程链表(就只有当前被唤醒线程)
                    thread_ptr -> tx_thread_ready_next =       thread_ptr;
                    thread_ptr -> tx_thread_ready_previous =   thread_ptr;

#if TX_MAX_PRIORITIES > 32

                    /* Calculate the index into the bit map array.  */
                    map_index =  priority/((UINT) 32);

                    /* Set the active bit to remember that the priority map has something set.  */
                    TX_DIV32_BIT_SET(priority, priority_bit)
                    _tx_thread_priority_map_active =  _tx_thread_priority_map_active | priority_bit;
#endif

                    /* Or in the thread's priority bit.  */
                    TX_MOD32_BIT_SET(priority, priority_bit)
```

```
                    _tx_thread_priority_maps[MAP_INDEX] = _tx_thread_priority_maps[MAP_INDEX] | priority_bit; // _tx_thread_priority_maps中，priority优先级对应的bit位设置为1，表示该优先级有就绪线程

                    /* Determine if this newly ready thread is the highest priority.  */
                    if (priority < _tx_thread_highest_priority) // 被唤醒线程的优先级高于被唤醒前的就绪线程的最高优先级
                    {

                        /* A new highest priority thread is present. */

                        /* Update the highest priority variable.  */
                        _tx_thread_highest_priority =  priority; // 更新就绪线程的最高优先级

                        /* Pickup the execute pointer. Since it is going to be referenced multiple
                           times, it is placed in a local variable.  */
                        execute_ptr =  _tx_thread_execute_ptr; // 获取_tx_thread_execute_ptr(_tx_thread_execute_ptr不一定是当前正在执行的线程，_tx_thread_execute_ptr只是调度程序选出来的下一个应该要执行的线程，优先级高或者抢占阈值高，但是因为禁止抢占等原因，_tx_thread_execute_ptr并不能立即执行或者还没来得及执行)

                        /* Determine if no thread is currently executing.  */
                        if (execute_ptr == TX_NULL) // _tx_thread_execute_ptr为0的话，表示之前没有就绪线程(当前唤醒操作可能是在中断服务程序里面执行)，那么当前线程就是下一个需要调度的线程
                        {

                            /* Simply setup the execute pointer.  */
                            _tx_thread_execute_ptr =  thread_ptr; // 设置被唤醒的线程为执行线程
                        }
                        else // 虽然被唤醒线程优先级高，但是被唤醒前调度器正要执行_tx_thread_execute_ptr线程，由于抢占阈值，被唤醒线程还得检查_tx_thread_execute_ptr是否启用了抢占，如果_tx_thread_execute_ptr启用了抢占，如果_tx_thread_execute_ptr抢占优先级高于被唤醒线程优先级，那么正在执行的_tx_thread_execute_ptr抢占被唤醒的高优先级线程...
```

```
161. 241                         {

162. 242

163. 243                             /* Another thread has been scheduled for execution. */

164. 244

165. 245                             /* Check to see if this is a higher priority thread and determine if preemption is allowed.  */

166. 246                             if (priority < execute_ptr -> tx_thread_preempt_threshold) // 正在执行的线程不能抢占被唤醒的线程

167. 247                             {

168. 248

169. 249 #ifndef TX_DISABLE_PREEMPTION_THRESHOLD

170. 250

171. 251                                 /* Determine if the preempted thread had preemption-threshold set.  */

172. 252                                 if (execute_ptr -> tx_thread_preempt_threshold != execute_ptr -> tx_thread_priority) // 如果正在执行的线程启用了抢占，启用了抢占的线程正在执行时，被抢占而换出cpu的话，那么需要标记一下，如果有高优先级线程退出的话，尽量尽快恢复被抢占的低优先级线程(ThreadX内核抢占阈值并不是在任何时候都有用，而是线程正在执行时以及恢复执行时才有用，线程创建及唤醒时，都是添加到就绪线程链表末尾，并不会用抢占阈值去抢占高优先级线程)

173. 253                                 {

174. 254

175. 255 #if TX_MAX_PRIORITIES > 32

176. 256

177. 257                                     /* Calculate the index into the bit map array.  */

178. 258                                     map_index =  (execute_ptr -> tx_thread_priority)/((UINT) 32);

179. 259

180. 260                                     /* Set the active bit to remember that the preempt map has something set.  */

181. 261                                     TX_DIV32_BIT_SET(execute_ptr -> tx_thread_priority, priority_bit)

182. 262                                     _tx_thread_preempted_map_active = _tx_thread_preempted_map_active | priority_bit;

183. 263 #endif

184. 264
```

185. 265 /* Remember that this thread was preempted by a thread above the thread's threshold. */

186. 266 TX_MOD32_BIT_SET(execute_ptr -> tx_thread_priority, priority_bit)

187. 267 _tx_thread_preempted_maps[MAP_INDEX] = _tx_thread_preempted_maps[MAP_INDEX] | priority_bit; // 设置启用了抢占的被抢占的线程的优先级在_tx_thread_preempted_map_active的对应bit位(被抢占出去的正在执行的线程还在就绪线程链表的表头，找到该优先级的就绪线程链表即可找到被抢占的线程；_tx_thread_preempted_map_active后续是从高优先级到低优先级来检查是否可以抢占最高优先级线程的，如果某个高优先级线程退出了，那么会检查_tx_thread_preempted_map_active标记的被抢占切换出去的线程的抢占阈值是否可以抢占下一个高优先级的线程；为什么要从高优先级开始检查呢，因为_tx_thread_preempted_map_active标记的高优先级线程的抢占阈值一定高于_tx_thread_preempted_map_active标记的低优先级线程的抢占阈值，_tx_thread_preempted_map_active高优先级线程要被标记的前提是该线程要在执行时被高优先级线程抢占，而高优先级线程要执行的前提是不能被_tx_thread_preempted_map_active标记的低优先级线程抢占)

188. 268 }

189. 269 #endif

190. 270

191. 271 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

192. 272

193. 273 /* Determine if the caller is an interrupt or from a thread. */

194. 274 if (TX_THREAD_GET_SYSTEM_STATE() == ((ULONG) 0))

195. 275 {

196. 276

197. 277 /* Caller is a thread, so this is a solicited preemption. */

198. 278 _tx_thread_performance_solicited_preemption_count++;

199. 279

200. 280 /* Increment the thread's solicited preemption counter. */

201. 281 execute_ptr -> tx_thread_performance_solicited_preemption_count++;

202. 282 }

203. 283 else

204. 284 {

205. 285

206. 286 if (TX_THREAD_GET_SYSTEM_STATE() < TX_INITIALIZE_IN_PROGRESS)

```
                        {

                            /* Caller is an interrupt, so this is an interrupt preemption.  */
_tx_thread_performance_interrupt_preemption_count++;

                            /* Increment the thread's interrupt preemption counter.  */
                            execute_ptr -> tx_thread_performance_interrupt_preemption_count++;
                        }
                    }

                    /* Remember the thread that preempted this thread.  */
                    execute_ptr -> tx_thread_performance_last_preempting_thread =  thread_ptr;

#endif

                    /* Yes, modify the execute thread pointer.  */
                    _tx_thread_execute_ptr =  thread_ptr; // 被唤醒线程的优先级最高，并且正在执行的线程不能抢占被唤醒的线程，设置被唤醒线程为执行线程

#ifndef TX_MISRA_ENABLE

                    /* If MISRA is not-enabled, insert a preemption and return in-line for performance.  */

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

                    /* Is the execute pointer different?  */
                    if (_tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] != _tx_thread_execute_ptr)
                    {
```

```

                    /* Move to next entry.  */
                    _tx_thread_performance__execute_log_index++;

                    /* Check for wrap condition.  */
                    if (_tx_thread_performance__execute_log_index >= TX_THREAD_EXECUTE_LOG_SIZE)
                    {

                        /* Set the index to the beginning.  */
                        _tx_thread_performance__execute_log_index =  ((UINT) 0);
                    }

                    /* Log the new execute pointer.  */
                    _tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] = _tx_thread_execute_ptr;
                }
#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 thread
                   resume 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, set the "next thread pointer" to NULL. This can
                       be used by the trace analysis tool to show idle system conditions.  */
                    entry_ptr -> tx_trace_buffer_entry_information_field_4 =  TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
                }
            }
#endif

            /* Restore interrupts.  */
            TX_RESTORE // 开中断

#ifdef TX_ENABLE_STACK_CHECKING

            /* Pickup the next execute pointer.  */
            thread_ptr =  _tx_thread_execute_ptr;

            /* Check this thread's stack.  */
            TX_THREAD_STACK_CHECK(thread_ptr)
#endif

            /* Now determine if preemption should take place. This is only possible if the current thread pointer is
               not the same as the execute thread pointer AND the system state and preempt disable flags are clear.  */
            TX_THREAD_SYSTEM_RETURN_CHECK(combined_flags) // 检查_tx_thread_preempt_disable是否禁止抢占以及是否在系统初始化阶段(禁止抢占或者还在系统初始化阶段，还得继续执行当前线程，不能进行线程切换)
            if (combined_flags == ((ULONG) 0))
            {

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

```

```
                                    /* There is another thread ready to run and will be scheduled upon return.  */
_tx_thread_performance_non_idle_return_count++;
#endif

                                    /* Preemption is needed - return to the system!  */
                                    _tx_thread_system_return(); // 非系统初始化阶段也没有禁止抢占，返回系统执行调度程序，调度被唤醒线程_tx_thread_execute_ptr
                                }

                                /* Return in-line when MISRA is not enabled.  */
                                return;
#endif
                            }
                        }
                    }
                }
                else // 被唤醒线程优先级所在就绪线程链表不为空，将被唤醒线程加入就绪线程链表即可(从下面代码可以看到线程加入的就绪线程链表末尾，也就是就绪线程链表是先进先出的，并不会因为抢占阈值而改变顺序)
                {

                    /* No, there are other threads at this priority already ready.  */

                    /* Just add this thread to the priority list.  */
                    tail_ptr =                                 head_ptr -> tx_thread_ready_previous;
                    tail_ptr -> tx_thread_ready_next =         thread_ptr;
                    head_ptr -> tx_thread_ready_previous =     thread_ptr;
                    thread_ptr -> tx_thread_ready_previous =   tail_ptr;
                    thread_ptr -> tx_thread_ready_next =       head_ptr;
                }
            }
```

```
                /* Else, delayed suspend flag was set.  */
                else // 延迟挂起标志位被设置，执行未执行完的挂起操作即可(例如线程等待互斥锁进入阻塞状态，别的线程调用挂起操作试图挂起等待互斥锁的线程，那么就会将阻塞的线程设置为等待挂起状态，阻塞的线程等到互斥锁后被唤醒，这里的else分支就会继续被挂起，不会真正唤醒线程，只是状态从阻塞变为挂起状态而已)
                {

                    /* Clear the delayed suspend flag and change the state.  */
                    thread_ptr -> tx_thread_delayed_suspend =  TX_FALSE;
                    thread_ptr -> tx_thread_state =            TX_SUSPENDED;
                }
            }
        }
        else // tx_thread_suspending被设置，线程挂起前还没调用_tx_thread_system_suspend的时候会设置tx_thread_suspending，说明有其他挂起操作还在进行中(tx_thread_suspending操作过程线程还没从就绪链表删除，另外线程结束时也调用挂起操作)
        {

            /* A resumption occurred in the middle of a previous thread suspension.  */

            /* Make sure the type of suspension under way is not a terminate or
               thread completion.  In either of these cases, do not void the
               interrupted suspension processing.  */
            if (thread_ptr -> tx_thread_state != TX_COMPLETED) // 检查线程是否处于TX_COMPLETED状态，线程是否已经结束，如果线程结束了，那么不需要唤醒线程
            {

                /* Make sure the thread isn't terminated.  */
                if (thread_ptr -> tx_thread_state != TX_TERMINATED) // 检查线程是否被终止，线程终止了也不需要唤醒线程
                {

                    /* No, now check to see if the delayed suspension flag is set.  */
```

```
                if (thread_ptr -> tx_thread_delayed_suspend == TX_FALSE)
                {

                    /* Clear the suspending flag.  */
                    thread_ptr -> tx_thread_suspending =   TX_FALSE;

                    /* Restore the state to ready.  */
                    thread_ptr -> tx_thread_state =        TX_READY;

                    /* Thread state change.  */
                    TX_THREAD_STATE_CHANGE(thread_ptr, TX_READY) // 唤醒线程(挂起中的线程没有从就绪链表删除，无需重新加入就绪链表)

                    /* Log the thread status change.  */
                    TX_EL_THREAD_STATUS_CHANGE_INSERT(thread_ptr, TX_READY)
                }
                else // tx_thread_suspending
                {

                    /* Clear the delayed suspend flag and change the state.  */
                    thread_ptr -> tx_thread_delayed_suspend =  TX_FALSE;
                    thread_ptr -> tx_thread_state =            TX_SUSPENDED;
                }

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

                /* Increment the total number of thread resumptions.  */
                _tx_thread_performance_resume_count++;

                /* Increment this thread's resume count.  */
                thread_ptr -> tx_thread_performance_resume_count++;
#endif
```

```
            }
        }
    }

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

    /* Is the execute pointer different?  */
    if (_tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] != _tx_thread_execute_ptr)
    {

        /* Move to next entry.  */
        _tx_thread_performance__execute_log_index++;

        /* Check for wrap condition.  */
        if (_tx_thread_performance__execute_log_index >= TX_THREAD_EXECUTE_LOG_SIZE)
        {

            /* Set the index to the beginning.  */
            _tx_thread_performance__execute_log_index =  ((UINT) 0);
        }

        /* Log the new execute pointer.  */
        _tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] = _tx_thread_execute_ptr;
    }
#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 thread
       resume 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, set the "next thread pointer" to NULL. This can
               be used by the trace analysis tool to show idle system conditions.  */
#ifdef TX_MISRA_ENABLE
            entry_ptr -> tx_trace_buffer_entry_info_4 = TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
#else
            entry_ptr -> tx_trace_buffer_entry_information_field_4 = TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
#endif
        }
    }
#endif

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

    /* Restore interrupts.  */
    TX_RESTORE

    /* Determine if a preemption condition is present.  */
    if (current_thread != _tx_thread_execute_ptr)
    {

```

```
#ifdef TX_ENABLE_STACK_CHECKING

        /* Pickup the next execute pointer.  */
        thread_ptr =  _tx_thread_execute_ptr;

        /* Check this thread's stack.  */
        TX_THREAD_STACK_CHECK(thread_ptr)
#endif

        /* Now determine if preemption should take place. This is only possible if the current thread pointer is
           not the same as the execute thread pointer AND the system state and preempt disable flags are clear.  */
        TX_THREAD_SYSTEM_RETURN_CHECK(combined_flags)
        if (combined_flags == ((ULONG) 0))
        {

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

            /* There is another thread ready to run and will be scheduled upon return.  */
            _tx_thread_performance_non_idle_return_count++;
#endif

            /* Preemption is needed - return to the system!  */
            _tx_thread_system_return();
        }
    }
}
```

5、线程恢复执行

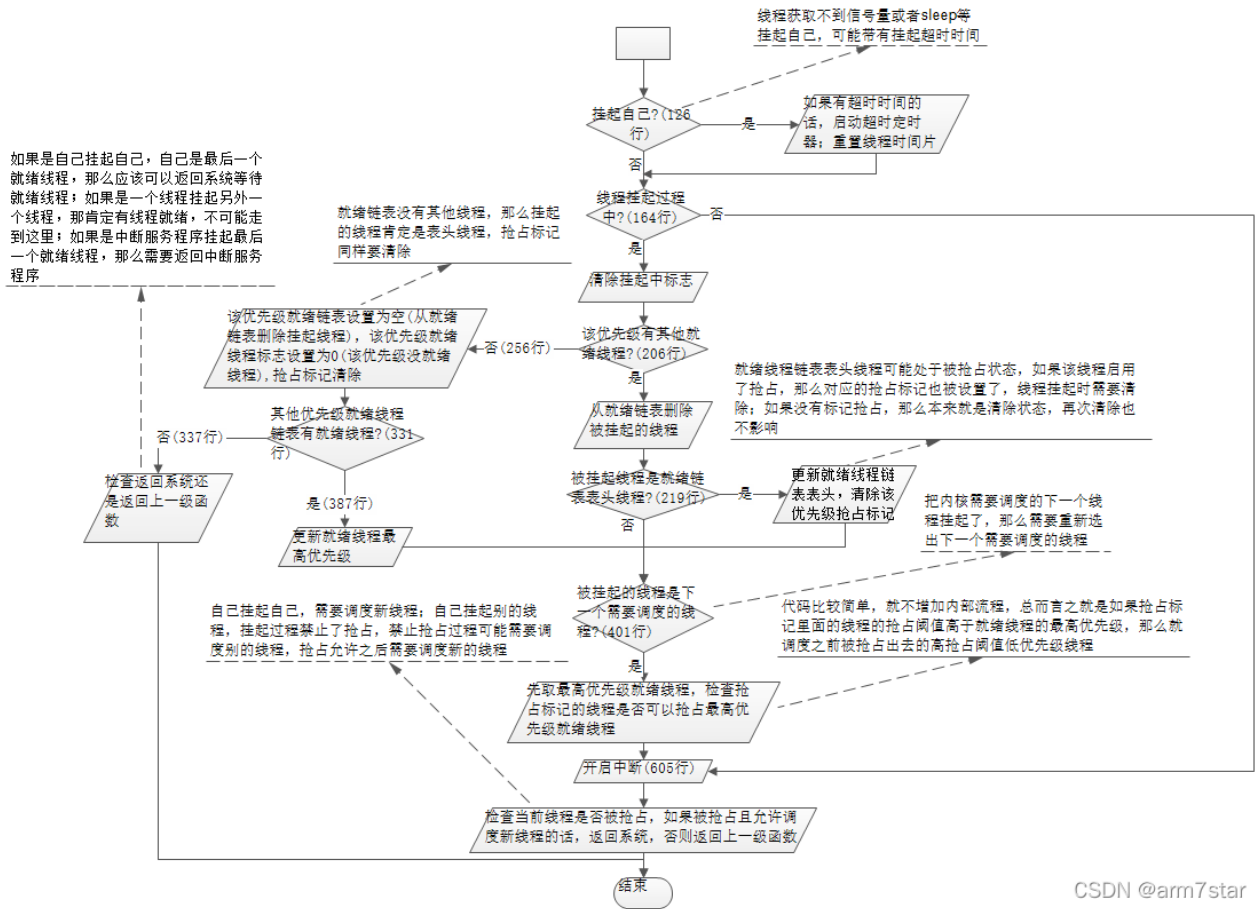
5.1、线程挂起过程恢复其他线程流程

阻塞/挂起线程恢复执行是在_tx_thread_system_resume里面实现的，前面已经介绍过了，这里只介绍就绪线程恢复执行；

就绪线程不能执行那么肯定有高优先级线程在执行或者同优先级线程还没执行完(时间片没有用尽，还没轮到自己执行)，时间片用尽前面介绍过，后面介绍线程退出时恢复线程的过程；

线程退出时，正常情况选择下一个高优先级线程执行即可，但是ThreadX启用了抢占，多了一个检查抢占的过程，就绪线程的优先级还需要跟之前被抢占线程的抢占阈值比较，以便能够恢复被抢占的启用了抢占的线程，启用了抢占的被抢占的动态优先级等于抢占阈值，按优先级并不能找到，因此要在_tx_thread_preempted_maps里面找有没有高优先级(抢占阈值高)的线程就绪；

ThreadX线程挂起一个线程的过程比较复杂，流程图大致如下所示：



5.2、线程挂起时恢复其他线程代码

线程挂起代码如下：

```
VOID  _tx_thread_system_suspend(TX_THREAD *thread_ptr)
#ifndef TX_NOT_INTERRUPTABLE
{

TX_INTERRUPT_SAVE_AREA

UINT            priority;
UINT            base_priority;
ULONG           priority_map;
ULONG           priority_bit;
ULONG           combined_flags;
TX_THREAD       *ready_next;
TX_THREAD       *ready_previous;
TX_THREAD       *current_thread;

#if TX_MAX_PRIORITIES > 32
UINT            map_index;
#endif

#ifndef TX_NO_TIMER
ULONG           timeout;
#endif

#ifdef TX_ENABLE_EVENT_TRACE
TX_TRACE_BUFFER_ENTRY       *entry_ptr;
ULONG                       time_stamp =  ((ULONG) 0);
#endif

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

```

```
#ifdef TX_ENABLE_STACK_CHECKING

    /* Check this thread's stack.  */
    TX_THREAD_STACK_CHECK(thread_ptr)
#endif

    /* Lockout interrupts while the thread is being suspended.  */
    TX_DISABLE

#ifndef TX_NO_TIMER

    /* Is the current thread suspending?  */
    if (thread_ptr == current_thread) // 挂起自己线程(可能是调用sleep或者等待信号量等导致的阻塞，可能需要超时唤醒)
    {

        /* Pickup the wait option.  */
        timeout =  thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks; // 获取超时时间，sleep或者等待信号量前会设置好tx_timer_internal_remaining_ticks

        /* Determine if an activation is needed.  */
        if (timeout != TX_NO_WAIT) // TX_NO_WAIT表示不等待，不用启动定时器
        {

            /* Make sure the suspension is not a wait-forever.  */
            if (timeout != TX_WAIT_FOREVER) // TX_WAIT_FOREVER表示没有超时时间，信号量等只有等到了才返回
            {

                /* Activate the thread timer with the timeout value setup in the caller.  */
                _tx_timer_system_activate(&(thread_ptr -> tx_thread_timer)); // 激活超时定时器(挂载到超时定时器链表等)
            }
```

```
        }

        /* Yes, reset time slice for current thread.  */
        _tx_timer_time_slice =  thread_ptr -> tx_thread_new_time_slice; // 重置当前线程的时间片(后面保存线程上下文的时候会把_tx_timer_time_slice保存到线程里面，下次唤醒后以新的时间片调度)
    }
#endif

    /* Decrease the preempt disabled count.  */
    _tx_thread_preempt_disable--;

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

    /* Increment the thread's suspend count.  */
    thread_ptr -> tx_thread_performance_suspend_count++;

    /* Increment the total number of thread suspensions.  */
    _tx_thread_performance_suspend_count++;
#endif

    /* Check to make sure the thread suspending flag is still set.  If not, it
       has already been resumed.  */
    if (thread_ptr -> tx_thread_suspending == TX_TRUE) // 挂起过程没被其他线程或者中断服务程序唤醒
    {

        /* Thread state change.  */
        TX_THREAD_STATE_CHANGE(thread_ptr, thread_ptr -> tx_thread_state)

        /* Log the thread status change.  */
        TX_EL_THREAD_STATUS_CHANGE_INSERT(thread_ptr, thread_ptr -> tx_thread_state)

```

```
#ifdef TX_ENABLE_EVENT_TRACE

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

        /* Log the thread status change.  */
        TX_TRACE_IN_LINE_INSERT(TX_TRACE_THREAD_SUSPEND, thread_ptr, thread_ptr -> tx_thread_state, TX_POINTER_TO_ULONG_CONVERT(&priority), TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr), TX_TRACE_INTERNAL_EVENTS)

#ifdef TX_ENABLE_EVENT_TRACE

        /* Save the time stamp for later comparison to verify that
           the event hasn't been overwritten by the time we have
           computed the next thread to execute.  */
        if (entry_ptr != TX_NULL)
        {

            /* Save time stamp.  */
            time_stamp =  entry_ptr -> tx_trace_buffer_entry_time_stamp;
        }
#endif

        /* Actually suspend this thread.  But first, clear the suspending flag. */
        thread_ptr -> tx_thread_suspending =  TX_FALSE; // tx_thread_suspending设置为TX_FALSE，这里已经关闭了中断，这个过程不会再被中断

        /* Pickup priority of thread.  */
        priority =  thread_ptr -> tx_thread_priority; // 被挂起线程的优先级

        /* Pickup the next ready thread pointer.  */
        ready_next =      thread_ptr -> tx_thread_ready_next;
```

```
        /* Determine if there are other threads at this priority that are
           ready.  */
        if (ready_next != thread_ptr) // 被挂起线程的就绪线程链表有其他线程
        {

            /* Yes, there are other threads at this priority ready.  */

            /* Pickup the previous ready thread pointer.  */
            ready_previous =  thread_ptr -> tx_thread_ready_previous;

            /* Just remove this thread from the priority list.  */ // 接下来几行代码将挂起线程从就绪线程链表删除
            ready_next -> tx_thread_ready_previous =    ready_previous;
            ready_previous -> tx_thread_ready_next =    ready_next;

            /* Determine if this is the head of the priority list.  */
            if (_tx_thread_priority_list[priority] == thread_ptr) // 检查被挂起线程是不是就绪线程链表的表头线程(是的话，需要更新就绪线程链表表头)
            {

                /* Update the head pointer of this priority list.  */
                _tx_thread_priority_list[priority] =  ready_next; // 更新就绪线程链表表头

#ifndef TX_DISABLE_PREEMPTION_THRESHOLD

#if TX_MAX_PRIORITIES > 32

                /* Calculate the index into the bit map array.  */
                map_index =  priority/((UINT) 32);
#endif

```

```
        /* Check for a thread preempted that had preemption threshold set.  */
        if (_tx_thread_preempted_maps[MAP_INDEX] != ((ULONG) 0)) // 前面有介绍，线程被执行的时候，线程仍在就绪线程链表表头，如果_tx_thread_preempted_maps被标记的话，需要清除该标记，因为现在的就绪线程链表的表头线程还没执行，不存在被抢占情况
        {

          /* Ensure that this thread's priority is clear in the preempt map.  */
          TX_MOD32_BIT_SET(priority, priority_bit)
          _tx_thread_preempted_maps[MAP_INDEX] = _tx_thread_preempted_maps[MAP_INDEX] & (~(priority_bit)); // 清除抢占标记

#if TX_MAX_PRIORITIES > 32

          /* Determine if there are any other bits set in this preempt map.  */
          if (_tx_thread_preempted_maps[MAP_INDEX] == ((ULONG) 0))
          {

            /* No, clear the active bit to signify this preempt map has nothing set.  */
            TX_DIV32_BIT_SET(priority, priority_bit)
            _tx_thread_preempted_map_active = _tx_thread_preempted_map_active & (~(priority_bit));
          }
#endif
        }
#endif
      }
    }
    else // 被挂起线程的下一个就绪线程指向自己(就绪线程链表里面没有其他线程了)
    {

      /* This is the only thread at this priority ready to run.  Set the head
```

```
178.                pointer to NULL.  */
179.             _tx_thread_priority_list[priority] =    TX_NULL; // 将被挂起线程的就绪线程链表清空即可
180. 
181. #if TX_MAX_PRIORITIES > 32
182. 
183.             /* Calculate the index into the bit map array.  */
184.             map_index =  priority/((UINT) 32);
185. #endif
186. 
187.             /* Clear this priority bit in the ready priority bit map.  */
188.             TX_MOD32_BIT_SET(priority, priority_bit)
189.             _tx_thread_priority_maps[MAP_INDEX] = _tx_thread_priority_maps[MAP_INDEX] & (~(priority_bit)); // 该优先级已经没有其他就绪线程了，清除对应的位
190. 
191. #if TX_MAX_PRIORITIES > 32
192. 
193.             /* Determine if there are any other bits set in this priority map.  */
194.             if (_tx_thread_priority_maps[MAP_INDEX] == ((ULONG) 0))
195.             {
196. 
197.                 /* No, clear the active bit to signify this priority map has nothing set.  */
198.                 TX_DIV32_BIT_SET(priority, priority_bit)
199.                 _tx_thread_priority_map_active =  _tx_thread_priority_map_active & (~(priority_bit));
200.             }
201. #endif
202. 
203. #ifndef TX_DISABLE_PREEMPTION_THRESHOLD
204. 
205.             /* Check for a thread preempted that had preemption-threshold set.  */
```

```
            if (_tx_thread_preempted_maps[MAP_INDEX] != ((ULONG) 0)) // _tx_thread_preempted_maps抢占标志位有被设置(不一定有标记被挂起的线程)
            {

                /* Ensure that this thread's priority is clear in the preempt map.  */
                TX_MOD32_BIT_SET(priority, priority_bit)
                _tx_thread_preempted_maps[MAP_INDEX] = _tx_thread_preempted_maps[MAP_INDEX] & (~(priority_bit)); // 清除被挂起线程优先级的抢占标记(被挂起线程有没有标记都要清除；有的话，线程已经挂起，没必要再调度)

#if TX_MAX_PRIORITIES > 32

                /* Determine if there are any other bits set in this preempt map.  */
                if (_tx_thread_preempted_maps[MAP_INDEX] == ((ULONG) 0))
                {

                    /* No, clear the active bit to signify this preempted map has nothing set.  */
                    TX_DIV32_BIT_SET(priority, priority_bit)
                    _tx_thread_preempted_map_active = _tx_thread_preempted_map_active & (~(priority_bit));
                }
#endif
            }
#endif

#if TX_MAX_PRIORITIES > 32

            /* Calculate the index to find the next highest priority thread ready for execution.  */
            priority_map =    _tx_thread_priority_map_active;

            /* Determine if there is anything.   */
            if (priority_map != ((ULONG) 0))
```

```
            {

                /* Calculate the lowest bit set in the priority map. */
                TX_LOWEST_SET_BIT_CALCULATE(priority_map, map_index)
            }

            /* Calculate the base priority as well.  */
            base_priority =  map_index * ((UINT) 32);
#else

            /* Setup the base priority to zero.  */
            base_priority =   ((UINT) 0);
#endif

            /* Setup working variable for the priority map.  */
            priority_map =    _tx_thread_priority_maps[MAP_INDEX]; // 检查是否有就绪线程(_tx_thread_priority_maps对应的位为1表示对应优先有就绪线程)

            /* Make a quick check for no other threads ready for execution.  */
            if (priority_map == ((ULONG) 0)) // 没有就绪线程
            {

                /* Nothing else is ready.  Set highest priority and execute thread
                   accordingly.  */
                _tx_thread_highest_priority =  ((UINT) TX_MAX_PRIORITIES); //最高就绪线程优先级TX_MAX_PRIORITIES
                _tx_thread_execute_ptr =       TX_NULL; // 需要调度的线程_tx_thread_execute_ptr设置为空，没有线程需要调度

#ifndef TX_MISRA_ENABLE

#ifdef TX_ENABLE_EVENT_TRACE

```

```
                /* Check that the event time stamp is unchanged.  A different
                   timestamp means that a later event wrote over the thread
                   suspend 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, set the "next thread pointer" to the new value of the
                           next thread to execute. This can be used by the trace analysis tool to keep
                           track of next thread execution.  */
                        entry_ptr -> tx_trace_buffer_entry_information_field_4 = 0;
                    }
                }
#endif

                /* Restore interrupts.  */
                TX_RESTORE

                /* Determine if preemption should take place. This is only possible if the current thread pointer is
                   not the same as the execute thread pointer AND the system state and preempt disable flags are clear.  */
                TX_THREAD_SYSTEM_RETURN_CHECK(combined_flags) // _tx_thread_preempt_disable等
                if (combined_flags == ((ULONG) 0)) // 检查是否禁止抢占是否在内核初始化过程，不是的话返回系统(没有就绪线程的话需要返回系统等待就绪线程)
                {

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
```

```

                    /* Yes, increment the return to idle return count.  */
                    _tx_thread_performance_idle_return_count++;
#endif

                    /* Preemption is needed - return to the system!  */
                    _tx_thread_system_return(); // 返回系统等待就绪线程
                }

                /* Return to caller.  */
                return;
#endif
            }
            else // 有就绪线程
            {

                /* Other threads at different priority levels are ready to run.  */

                /* Calculate the lowest bit set in the priority map. */
                TX_LOWEST_SET_BIT_CALCULATE(priority_map, priority_bit)

                /* Setup the next highest priority variable.  */
                _tx_thread_highest_priority =  base_priority + ((UINT) priority_bit); // 更新就绪线程最高优先级(被挂起线程优先级所在就绪线程链表没有其他线程)
            }
        }

        /* Determine if the suspending thread is the thread designated to execute.  */
        if (thread_ptr == _tx_thread_execute_ptr) // 挂起正在执行的线程(没有其他就绪线程的情况，前面已经处理过了，这里应该就是有其他就绪线程的情况，需要调度别的就绪线程；挂起不在执行的线程不会导致线程调度)
        {
```

```

            /* Pickup the highest priority thread to execute.  */
            _tx_thread_execute_ptr = _tx_thread_priority_list[_tx_thread_highest_priority]; // _tx_thread_execute_ptr指向最高优先级就绪线程

#ifndef TX_DISABLE_PREEMPTION_THRESHOLD

            /* Determine if a previous thread with preemption-threshold was preempted.  */
#if TX_MAX_PRIORITIES > 32
            if (_tx_thread_preempted_map_active != ((ULONG) 0))
#else
            if (_tx_thread_preempted_maps[MAP_INDEX] != ((ULONG) 0)) // 检查是否有启用抢占的线程被抢占
#endif
            {

                /* Yes, there was a thread preempted when it was using preemption-threshold.  */

                /* Disable preemption.  */
                _tx_thread_preempt_disable++; // 禁止抢占(后面开中断让中断得到及时处理，禁止抢占避免当前过程被其他线程中断)

                /* Restore interrupts.  */
                TX_RESTORE

                /* Interrupts are enabled briefly here to keep the interrupt
                   lockout time deterministic.  */

                /* Disable interrupts again.  */
                TX_DISABLE

                /* Decrement the preemption disable variable.  */
```

```
                _tx_thread_preempt_disable--;

                /* Calculate the thread with preemption threshold set that
                   was interrupted by a thread above the preemption level.  */

#if TX_MAX_PRIORITIES > 32

                /* Calculate the index to find the next highest priority thread ready for execution.  */
                priority_map =    _tx_thread_preempted_map_active;

                /* Calculate the lowest bit set in the priority map. */
                TX_LOWEST_SET_BIT_CALCULATE(priority_map, map_index)

                /* Calculate the base priority as well.  */
                base_priority =  map_index * ((UINT) 32);
#else

                /* Setup the base priority to zero.  */
                base_priority =   ((UINT) 0);
#endif

                /* Setup temporary preempted map.  */
                priority_map =  _tx_thread_preempted_maps[MAP_INDEX];

                /* Calculate the lowest bit set in the priority map. */
                TX_LOWEST_SET_BIT_CALCULATE(priority_map, priority_bit) // 获取最高优先级的抢占线程的标记位(前面讲过了，_tx_thread_preempted_maps里面的高优先级线程的优先级高于低优先级线程的抢占阈值，要调度的话也只可能是_tx_thread_preempted_maps里面高优先级线程)

                /* Setup the highest priority preempted thread.  */
                priority =  base_priority + ((UINT) priority_bit); // 获取_tx_thread_preempted_maps标记的最高优先级线程的优先级
```

```

                /* Determine if the next highest priority thread is above the highest priority threshold value.  */
                if (_tx_thread_highest_priority >= (_tx_thread_priority_list[priority] -> tx_thread_preempt_threshold)) // 最高优先级就绪线程的优先级不高于_tx_thread_preempted_maps被抢占线程的抢占阈值，_tx_thread_preempted_maps里面的高优先级线程抢占下一个就绪的最高优先级线程
                {

                    /* Thread not allowed to execute until earlier preempted thread finishes or lowers its
                       preemption-threshold.  */
                    _tx_thread_execute_ptr = _tx_thread_priority_list[priority]; // 获取_tx_thread_preempted_maps标记的被抢占的线程(被抢占的线程还在就绪线程链表_tx_thread_priority_list的表头)

                    /* Clear the corresponding bit in the preempted map, since the preemption has been restored.  */
                    TX_MOD32_BIT_SET(priority, priority_bit)
                    _tx_thread_preempted_maps[MAP_INDEX] = _tx_thread_preempted_maps[MAP_INDEX] & (~(priority_bit)); // 清除_tx_thread_preempted_maps里面的抢占标记

#if TX_MAX_PRIORITIES > 32

                    /* Determine if there are any other bits set in this preempt map.  */
                    if (_tx_thread_preempted_maps[MAP_INDEX] == ((ULONG) 0))
                    {

                        /* No, clear the active bit to signify this preempt map has nothing set.  */
                        TX_DIV32_BIT_SET(priority, priority_bit)
                        _tx_thread_preempted_map_active = _tx_thread_preempted_map_active & (~(priority_bit));
                    }
#endif
                }
```

```
            }
#endif

#ifndef TX_MISRA_ENABLE

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

            /* Is the execute pointer different?  */
            if (_tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] != _tx_thread_execute_ptr)
            {

                /* Move to next entry.  */
                _tx_thread_performance__execute_log_index++;

                /* Check for wrap condition.  */
                if (_tx_thread_performance__execute_log_index >= TX_THREAD_EXECUTE_LOG_SIZE)
                {

                    /* Set the index to the beginning.  */
                    _tx_thread_performance__execute_log_index =  ((UINT) 0);
                }

                /* Log the new execute pointer.  */
_tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] = _tx_thread_execute_ptr;
            }
#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 thread
               suspend 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, set the "next thread pointer" to the new value of the
                       next thread to execute. This can be used by the trace analysis tool to keep
                       track of next thread execution.  */
                    entry_ptr -> tx_trace_buffer_entry_information_field_4 = TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
                }
            }
#endif

            /* Restore interrupts.  */
            TX_RESTORE

            /* Determine if preemption should take place. This is only possible if the current thread pointer is
               not the same as the execute thread pointer AND the system state and preempt disable flags are clear.  */
            TX_THREAD_SYSTEM_RETURN_CHECK(combined_flags)
            if (combined_flags == ((ULONG) 0)) // 与之前一样，检查是返回系统调度线程还是返回
            {

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

```

```
                /* No, there is another thread ready to run and will be scheduled upon return.  */
                _tx_thread_performance_non_idle_return_count++;
#endif

                /* Preemption is needed - return to the system!  */
                _tx_thread_system_return();
            }

            /* Return to caller.  */
            return; // 返回
#endif
        }

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

        /* Is the execute pointer different?  */
        if (_tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] != _tx_thread_execute_ptr)
        {

            /* Move to next entry.  */
            _tx_thread_performance__execute_log_index++;

            /* Check for wrap condition.  */
            if (_tx_thread_performance__execute_log_index >= TX_THREAD_EXECUTE_LOG_SIZE)
            {

                /* Set the index to the beginning.  */
                _tx_thread_performance__execute_log_index =  ((UINT) 0);
            }

```

```
            /* Log the new execute pointer.  */
            _tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] = _tx_thread_execute_ptr;
        }
#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 thread
            suspend 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, set the "next thread pointer" to the new value of the
                   next thread to execute. This can be used by the trace analysis tool to keep
                   track of next thread execution.  */
#ifdef TX_MISRA_ENABLE
                entry_ptr -> tx_trace_buffer_entry_info_4 = TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
#else
                entry_ptr -> tx_trace_buffer_entry_information_field_4 = TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
#endif
            }
        }
#endif
    }
```

```
    /* Restore interrupts.  */
    TX_RESTORE
    /* Determine if a preemption condition is present.  */
    if (current_thread != _tx_thread_execute_ptr)
    {
#ifdef TX_ENABLE_STACK_CHECKING
        /* Pickup the next execute pointer.  */
        thread_ptr =  _tx_thread_execute_ptr;
        /* Check this thread's stack.  */
        TX_THREAD_STACK_CHECK(thread_ptr)
#endif
        /* Determine if preemption should take place. This is only possible if the current thread pointer is
           not the same as the execute thread pointer AND the system state and preempt disable flags are clear.  */
        TX_THREAD_SYSTEM_RETURN_CHECK(combined_flags)
        if (combined_flags == ((ULONG) 0))
        {
#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
            /* Determine if an idle system return is present.  */
            if (_tx_thread_execute_ptr == TX_NULL)
            {
                /* Yes, increment the return to idle return count.  */
                _tx_thread_performance_idle_return_count++;
```

```
            }
            else
            {

                /* No, there is another thread ready to run and will be scheduled upon return.  */
                _tx_thread_performance_non_idle_return_count++;
            }
#endif

            /* Preemption is needed - return to the system!  */
            _tx_thread_system_return();
        }
    }

    /* Return to caller.  */
    return;
}
```