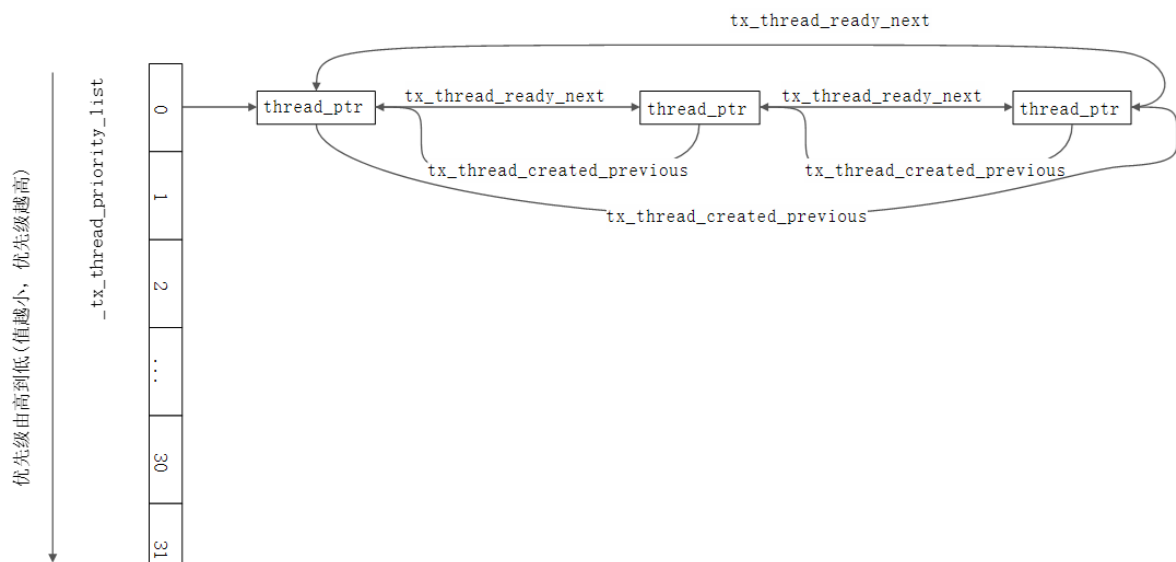


(163条消息) ThreadX内核源码分析 - 优先级及抢占阈值抢占(arm)_arm7star的博客-CSDN博客_threadx 优先级位图

 blog.csdn.net/arm7star/article/details/123001013

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)。



CSDN @arm7star

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：

```
1. 081 VOID _tx_thread_system_resume(TX_THREAD *thread_ptr)
2. 082 #ifndef TX_NOT_INTERRUPTABLE
3. 083 {
4. 084
5. 085 TX_INTERRUPT_SAVE_AREA
6. 086
7. 087 UINT          priority;
8. 088 ULONG          priority_bit;
9. 089 TX_THREAD      *head_ptr;
10. 090 TX_THREAD      *tail_ptr;
11. 091 TX_THREAD      *execute_ptr;
12. 092 TX_THREAD      *current_thread;
13. 093 ULONG          combined_flags;
14. 094
15. 095 #ifdef TX_ENABLE_EVENT_TRACE
16. 096 TX_TRACE_BUFFER_ENTRY *entry_ptr;
17. 097 ULONG                  time_stamp = ((ULONG) 0);
18. 098 #endif
19. 099
20. 100 #if TX_MAX_PRIORITIES > 32
21. 101 UINT          map_index;
22. 102 #endif
23. 103
24. 104
25. 105 #ifdef TX_ENABLE_STACK_CHECKING
26. 106
27. 107     /* Check this thread's stack. */
28. 108     TX_THREAD_STACK_CHECK(thread_ptr)
29. 109 #endif
30. 110
31. 111     /* Lockout interrupts while the thread is being resumed. */
```

```

32. 112     TX_DISABLE

33. 113

34. 114 #ifndef TX_NO_TIMER

35. 115

36. 116     /* Deactivate the timeout timer if necessary. */

37. 117     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

38. 118     {

39. 119

40. 120         /* Deactivate the thread's timeout timer. */

41. 121         _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)

42. 122     }

43. 123     else

44. 124     {

45. 125

46. 126         /* Clear the remaining time to ensure timer doesn't get activated. */

47. 127         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)

48. 128     }

49. 129 #endif

50. 130

51. 131 #ifdef TX_ENABLE_EVENT_TRACE

52. 132

53. 133     /* If trace is enabled, save the current event pointer. */

54. 134     entry_ptr = _tx_trace_buffer_current_ptr;

55. 135 #endif

56. 136

```

```

57. 137      /* Log the thread status change. */

58. 138      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)

59. 139

60. 140 #ifdef TX_ENABLE_EVENT_TRACE

61. 141

62. 142      /* Save the time stamp for later comparison to verify that

63. 143          the event hasn't been overwritten by the time we have

64. 144          computed the next thread to execute. */

65. 145      if (entry_ptr != TX_NULL)

66. 146      {

67. 147

68. 148          /* Save time stamp. */

69. 149          time_stamp = entry_ptr -> tx_trace_buffer_entry_time_stamp;

70. 150      }

71. 151 #endif

72. 152

73. 153      /* Decrease the preempt disabled count. */

74. 154      _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)

75. 155

76. 156      /* Determine if the thread is in the process of suspending. If so, the
    thread

77. 157          control block is already on the linked list so nothing needs to be done.
    */

78. 158      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)

79. 159      {

80. 160

81. 161          /* Thread is not in the process of suspending. Now check to make sure
    the thread

82. 162              has not already been resumed. */

```

```

83. 163         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)

84. 164         {

85. 165

86. 166             /* No, now check to see if the delayed suspension flag is set. */

87. 167             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)

88. 168         {

89. 169

90. 170             /* Resume the thread! */

91. 171

92. 172             /* Make this thread ready. */

93. 173

94. 174             /* Change the state to ready. */

95. 175             thread_ptr -> tx_thread_state = TX_READY; // 线程状态变为就绪状态

96. 176

97. 177             /* Pickup priority of thread. */

98. 178             priority = thread_ptr -> tx_thread_priority;

99. 179

100. 180             /* Thread state change. */

101. 181             TX_THREAD_STATE_CHANGE(thread_ptr, TX_READY)

102. 182

103. 183             /* Log the thread status change. */

104. 184             TX_EL_THREAD_STATUS_CHANGE_INSERT(thread_ptr, TX_READY)

105. 185

106. 186 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

107. 187

108. 188             /* Increment the total number of thread resumptions. */

109. 189             _tx_thread_performance_resume_count++;

110. 190

```

```

111. 191          /* Increment this thread's resume count.  */
112. 192          thread_ptr -> tx_thread_performance_resume_count++;
113. 193 #endif
114. 194
115. 195          /* Determine if there are other threads at this priority that
116. 196             are
117. 197             ready.  */
118. 198          head_ptr = _tx_thread_priority_list[priority]; // 线程所在就绪
线程链表
119. 199          {
120. 200
121. 201          /* First thread at this priority ready.  Add to the front of
122. 202             the list.  */
123. 203          _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)
124. 204          thread_ptr -> tx_thread_ready_next = thread_ptr;
125. 205          thread_ptr -> tx_thread_ready_previous = thread_ptr;
126. 206 #if TX_MAX_PRIORITIES > 32
127. 207
128. 208          /* Calculate the index into the bit map array.  */
129. 209          map_index = priority/((UINT) 32);
130. 210
131. 211          /* Set the active bit to remember that the priority map has
132. 212             something set.  */
133. 213          TX_DIV32_BIT_SET(priority, priority_bit)
134. 214          _tx_thread_priority_map_active =
_tx_thread_priority_map_active | priority_bit;
135. 215
136. 216          /* Or in the thread's priority bit.  */
137. 217          TX_MOD32_BIT_SET(priority, priority_bit)

```



```

138. 218                _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

139. 219

140. 220                /* Determine if this newly ready thread is the highest
    priority. */

141. 221                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

142. 222                {

143. 223

144. 224                /* A new highest priority thread is present. */

145. 225

146. 226                /* Update the highest priority variable. */

147. 227                _tx_thread_highest_priority = priority; // 更新就绪线程
    的最高优先级

148. 228

149. 229                /* Pickup the execute pointer. Since it is going to be
    referenced multiple

150. 230                times, it is placed in a local variable. */

151. 231                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)

152. 232

153. 233                /* Determine if no thread is currently executing. */

154. 234                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

155. 235                {

156. 236

157. 237                /* Simply setup the execute pointer. */

158. 238                _tx_thread_execute_ptr = thread_ptr; // 设置被唤醒
    的线程为执行线程

159. 239                }

160. 240                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)

```

```

207. 287                                {
208. 288
209. 289                                /* Caller is an interrupt, so this is an
interrupt preemption. */
210. 290                                _tx_thread_performance_interrupt_preemption_count++;
211. 291
212. 292                                /* Increment the thread's interrupt
preemption counter. */
213. 293                                execute_ptr ->
tx_thread_performance_interrupt_preemption_count++;
214. 294                                }
215. 295                                }
216. 296
217. 297                                /* Remember the thread that preempted this
thread. */
218. 298                                execute_ptr ->
tx_thread_performance_last_preempting_thread = thread_ptr;
219. 299
220. 300 #endif
221. 301
222. 302                                /* Yes, modify the execute thread pointer. */
223. 303                                _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
224. 304
225. 305 #ifndef TX_MISRA_ENABLE
226. 306
227. 307                                /* If MISRA is not-enabled, insert a preemption
and return in-line for performance. */
228. 308
229. 309 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
230. 310
231. 311                                /* Is the execute pointer different? */
232. 312                                if
(_tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] !=
_tx_thread_execute_ptr)
233. 313                                {

```

```

234. 314
235. 315                                /* Move to next entry.  */
236. 316                                _tx_thread_performance__execute_log_index++;
237. 317
238. 318                                /* Check for wrap condition.  */
239. 319                                if
    (_tx_thread_performance__execute_log_index >= TX_THREAD_EXECUTE_LOG_SIZE)
240. 320                                {
241. 321
242. 322                                /* Set the index to the beginning.  */
243. 323                                _tx_thread_performance__execute_log_index = ((UINT) 0);
244. 324                                }
245. 325
246. 326                                /* Log the new execute pointer.  */
247. 327                                _tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] =
    _tx_thread_execute_ptr;
248. 328                                }
249. 329 #endif
250. 330
251. 331 #ifdef TX_ENABLE_EVENT_TRACE
252. 332
253. 333                                /* Check that the event time stamp is unchanged.
    A different
254. 334                                timestamp means that a later event wrote over
    the thread
255. 335                                resume event. In that case, do nothing here.
    */
256. 336                                if (entry_ptr != TX_NULL)
257. 337                                {
258. 338
259. 339                                /* Is the timestamp the same?  */
260. 340                                if (time_stamp == entry_ptr ->
    tx_trace_buffer_entry_time_stamp)
261. 341                                {

```

```

262. 342
263. 343                                     /* Timestamp is the same, set the "next
thread pointer" to NULL. This can
264. 344                                     be used by the trace analysis tool to
show idle system conditions. */
265. 345                                     entry_ptr ->
tx_trace_buffer_entry_information_field_4 =
TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
266. 346                                     }
267. 347                                     }
268. 348 #endif
269. 349
270. 350                                     /* Restore interrupts. */
271. 351                                     TX_RESTORE // 开中断
272. 352
273. 353 #ifdef TX_ENABLE_STACK_CHECKING
274. 354
275. 355                                     /* Pickup the next execute pointer. */
276. 356                                     thread_ptr = _tx_thread_execute_ptr;
277. 357
278. 358                                     /* Check this thread's stack. */
279. 359                                     TX_THREAD_STACK_CHECK(thread_ptr)
280. 360 #endif
281. 361
282. 362                                     /* Now determine if preemption should take
place. This is only possible if the current thread pointer is
283. 363                                     not the same as the execute thread pointer
AND the system state and preempt disable flags are clear. */
284. 364                                     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)
285. 365                                     if (combined_flags == ((ULONG) 0))
286. 366                                     {
287. 367
288. 368 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
289. 369

```

```

290. 370                                     /* There is another thread ready to run and
      will be scheduled upon return. */

291. 371
      _tx_thread_performance_non_idle_return_count++;

292. 372 #endif

293. 373

294. 374                                     /* Preemption is needed - return to the
      system! */

295. 375                                     _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

296. 376                                     }

297. 377

298. 378                                     /* Return in-line when MISRA is not enabled. */

299. 379                                     return;

300. 380 #endif

301. 381                                     }

302. 382                                     }

303. 383                                     }

304. 384                                     }

305. 385                                     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)

306. 386                                     {

307. 387

308. 388                                     /* No, there are other threads at this priority already
      ready. */

309. 389

310. 390                                     /* Just add this thread to the priority list. */

311. 391                                     tail_ptr =                                     head_ptr ->
      tx_thread_ready_previous;

312. 392                                     tail_ptr -> tx_thread_ready_next =                                     thread_ptr;

313. 393                                     head_ptr -> tx_thread_ready_previous =                                     thread_ptr;

314. 394                                     thread_ptr -> tx_thread_ready_previous =                                     tail_ptr;

315. 395                                     thread_ptr -> tx_thread_ready_next =                                     head_ptr;

316. 396                                     }

317. 397                                     }

```

```

318. 398
319. 399          /* Else, delayed suspend flag was set.  */

320. 400          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)

321. 401          {

322. 402

323. 403          /* Clear the delayed suspend flag and change the state.  */

324. 404          thread_ptr -> tx_thread_delayed_suspend = TX_FALSE;

325. 405          thread_ptr -> tx_thread_state = TX_SUSPENDED;

326. 406          }

327. 407      }

328. 408  }

329. 409      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)

330. 410      {

331. 411

332. 412          /* A resumption occurred in the middle of a previous thread suspension.
            */

333. 413

334. 414          /* Make sure the type of suspension under way is not a terminate or

335. 415          thread completion. In either of these cases, do not void the

336. 416          interrupted suspension processing.  */

337. 417          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

338. 418          {

339. 419

340. 420          /* Make sure the thread isn't terminated.  */

341. 421          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

342. 422          {

343. 423

344. 424          /* No, now check to see if the delayed suspension flag is set.
            */

```



```

345. 425         if (thread_ptr -> tx_thread_delayed_suspend == TX_FALSE)
346. 426         {
347. 427
348. 428             /* Clear the suspending flag. */
349. 429             thread_ptr -> tx_thread_suspending = TX_FALSE;
350. 430
351. 431             /* Restore the state to ready. */
352. 432             thread_ptr -> tx_thread_state = TX_READY;
353. 433
354. 434             /* Thread state change. */
355. 435             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)
356. 436
357. 437             /* Log the thread status change. */
358. 438             TX_EL_THREAD_STATUS_CHANGE_INSERT(thread_ptr, TX_READY)
359. 439         }
360. 440     else // tx_thread_suspending
361. 441     {
362. 442
363. 443         /* Clear the delayed suspend flag and change the state. */
364. 444         thread_ptr -> tx_thread_delayed_suspend = TX_FALSE;
365. 445         thread_ptr -> tx_thread_state = TX_SUSPENDED;
366. 446     }
367. 447
368. 448 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
369. 449
370. 450         /* Increment the total number of thread resumptions. */
371. 451         _tx_thread_performance_resume_count++;
372. 452
373. 453         /* Increment this thread's resume count. */
374. 454         thread_ptr -> tx_thread_performance_resume_count++;
375. 455 #endif

```

```

376. 456         }
377. 457     }
378. 458 }
379. 459
380. 460 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
381. 461
382. 462     /* Is the execute pointer different? */
383. 463     if
        (_tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] !=
        _tx_thread_execute_ptr)
384. 464     {
385. 465
386. 466         /* Move to next entry. */
387. 467         _tx_thread_performance__execute_log_index++;
388. 468
389. 469         /* Check for wrap condition. */
390. 470         if (_tx_thread_performance__execute_log_index >=
            TX_THREAD_EXECUTE_LOG_SIZE)
391. 471         {
392. 472
393. 473             /* Set the index to the beginning. */
394. 474             _tx_thread_performance__execute_log_index = ((UINT) 0);
395. 475         }
396. 476
397. 477         /* Log the new execute pointer. */
398. 478         _tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] =
            _tx_thread_execute_ptr;
399. 479     }
400. 480 #endif
401. 481
402. 482 #ifdef TX_ENABLE_EVENT_TRACE
403. 483
404. 484     /* Check that the event time stamp is unchanged. A different

```

```

405. 485         timestamp means that a later event wrote over the thread
406. 486         resume event. In that case, do nothing here.  */
407. 487     if (entry_ptr != TX_NULL)
408. 488     {
409. 489
410. 490         /* Is the timestamp the same?  */
411. 491         if (time_stamp == entry_ptr -> tx_trace_buffer_entry_time_stamp)
412. 492         {
413. 493
414. 494             /* Timestamp is the same, set the "next thread pointer" to NULL.
               This can
415. 495             be used by the trace analysis tool to show idle system
               conditions.  */
416. 496 #ifdef TX_MISRA_ENABLE
417. 497         entry_ptr -> tx_trace_buffer_entry_info_4 =
               TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
418. 498 #else
419. 499         entry_ptr -> tx_trace_buffer_entry_information_field_4 =
               TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
420. 500 #endif
421. 501     }
422. 502 }
423. 503 #endif
424. 504
425. 505     /* Pickup thread pointer.  */
426. 506     TX_THREAD_GET_CURRENT(current_thread)
427. 507
428. 508     /* Restore interrupts.  */
429. 509     TX_RESTORE
430. 510
431. 511     /* Determine if a preemption condition is present.  */
432. 512     if (current_thread != _tx_thread_execute_ptr)
433. 513     {
434. 514

```

```

435. 515 #ifdef TX_ENABLE_STACK_CHECKING

436. 516

437. 517     /* Pickup the next execute pointer. */

438. 518     thread_ptr = _tx_thread_execute_ptr;

439. 519

440. 520     /* Check this thread's stack. */

441. 521     TX_THREAD_STACK_CHECK(thread_ptr)

442. 522 #endif

443. 523

444. 524     /* Now determine if preemption should take place. This is only possible
        if the current thread pointer is

445. 525     not the same as the execute thread pointer AND the system state and
        preempt disable flags are clear. */

446. 526     TX_THREAD_SYSTEM_RETURN_CHECK(combined_flags)

447. 527     if (combined_flags == ((ULONG) 0))

448. 528     {

449. 529

450. 530 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

451. 531

452. 532     /* There is another thread ready to run and will be scheduled upon
        return. */

453. 533     _tx_thread_performance_non_idle_return_count++;

454. 534 #endif

455. 535

456. 536     /* Preemption is needed - return to the system! */

457. 537     _tx_thread_system_return();

458. 538 }

459. 539 }

460. 540 }

```



5、线程恢复执行

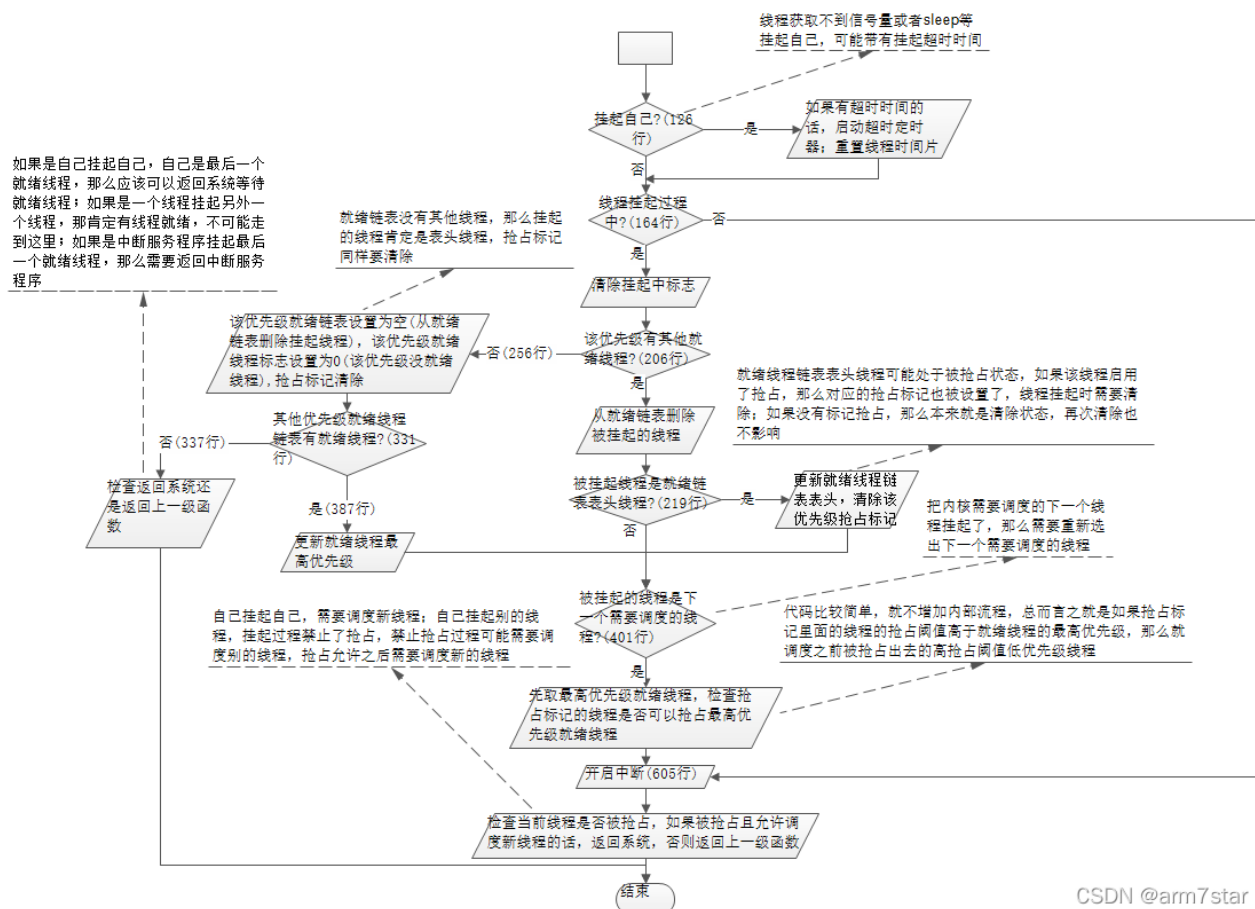
5.1、线程挂起过程恢复其他线程流程

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：

```
1. 083 VOID _tx_thread_system_suspend(TX_THREAD *thread_ptr)

2. 084 #ifndef TX_NOT_INTERRUPTABLE

3. 085 {

4. 086

5. 087 TX_INTERRUPT_SAVE_AREA

6. 088

7. 089 UINT          priority;

8. 090 UINT          base_priority;

9. 091 ULONG         priority_map;

10. 092 ULONG        priority_bit;

11. 093 ULONG        combined_flags;

12. 094 TX_THREAD    *ready_next;

13. 095 TX_THREAD    *ready_previous;

14. 096 TX_THREAD    *current_thread;

15. 097

16. 098 #if TX_MAX_PRIORITIES > 32

17. 099 UINT          map_index;

18. 100 #endif

19. 101

20. 102 #ifndef TX_NO_TIMER

21. 103 ULONG         timeout;

22. 104 #endif

23. 105

24. 106 #ifdef TX_ENABLE_EVENT_TRACE

25. 107 TX_TRACE_BUFFER_ENTRY    *entry_ptr;

26. 108 ULONG                    time_stamp = ((ULONG) 0);

27. 109 #endif

28. 110

29. 111     /* Pickup thread pointer. */

30. 112     TX_THREAD_GET_CURRENT(current_thread)

31. 113
```

```
32. 114 #ifdef TX_ENABLE_STACK_CHECKING

33. 115

34. 116     /* Check this thread's stack. */

35. 117     TX_THREAD_STACK_CHECK(thread_ptr)

36. 118 #endif

37. 119

38. 120     /* Lockout interrupts while the thread is being suspended. */

39. 121     TX_DISABLE

40. 122

41. 123 #ifndef TX_NO_TIMER

42. 124

43. 125     /* Is the current thread suspending? */

44. 126     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)

45. 127     {

46. 128

47. 129         /* Pickup the wait option. */

48. 130         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

49. 131

50. 132         /* Determine if an activation is needed. */

51. 133         if (timeout != TX_NO_WAIT) // TX_NO_WAIT表示不等待，不用启动定时器

52. 134         {

53. 135

54. 136             /* Make sure the suspension is not a wait-forever. */

55. 137             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

56. 138             {

57. 139

58. 140                 /* Activate the thread timer with the timeout value setup in the
                    caller. */

59. 141                 _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)

60. 142             }
```

```

61. 143     }

62. 144

63. 145     /* Yes, reset time slice for current thread. */

64. 146     _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)

65. 147     }

66. 148 #endif

67. 149

68. 150     /* Decrease the preempt disabled count. */

69. 151     _tx_thread_preempt_disable--;

70. 152

71. 153 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

72. 154

73. 155     /* Increment the thread's suspend count. */

74. 156     thread_ptr -> tx_thread_performance_suspend_count++;

75. 157

76. 158     /* Increment the total number of thread suspensions. */

77. 159     _tx_thread_performance_suspend_count++;

78. 160 #endif

79. 161

80. 162     /* Check to make sure the thread suspending flag is still set. If not, it
81. 163        has already been resumed. */

82. 164     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

83. 165     {

84. 166

85. 167         /* Thread state change. */

86. 168         TX_THREAD_STATE_CHANGE(thread_ptr, thread_ptr -> tx_thread_state)

87. 169

88. 170         /* Log the thread status change. */

89. 171         TX_EL_THREAD_STATUS_CHANGE_INSERT(thread_ptr, thread_ptr ->
    tx_thread_state)

90. 172

```



```

91. 173 #ifdef TX_ENABLE_EVENT_TRACE

92. 174

93. 175         /* If trace is enabled, save the current event pointer.  */

94. 176         entry_ptr = _tx_trace_buffer_current_ptr;

95. 177 #endif

96. 178

97. 179         /* Log the thread status change.  */

98. 180         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)

99. 181

100. 182 #ifdef TX_ENABLE_EVENT_TRACE

101. 183

102. 184         /* Save the time stamp for later comparison to verify that

103. 185         the event hasn't been overwritten by the time we have

104. 186         computed the next thread to execute.  */

105. 187         if (entry_ptr != TX_NULL)

106. 188         {

107. 189

108. 190                 /* Save time stamp.  */

109. 191                 time_stamp = entry_ptr -> tx_trace_buffer_entry_time_stamp;

110. 192         }

111. 193 #endif

112. 194

113. 195         /* Actually suspend this thread.  But first, clear the suspending flag.
*/

114. 196         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

115. 197

116. 198         /* Pickup priority of thread.  */

117. 199         priority = thread_ptr -> tx_thread_priority; // 被挂起线程的优先级

118. 200

119. 201         /* Pickup the next ready thread pointer.  */

120. 202         ready_next = thread_ptr -> tx_thread_ready_next;

```

```

121. 203
122. 204         /* Determine if there are other threads at this priority that are
123. 205             ready. */
124. 206         if (ready_next != thread_ptr) // 被挂起线程的就绪线程链表有其他线程
125. 207         {
126. 208
127. 209             /* Yes, there are other threads at this priority ready. */
128. 210
129. 211             /* Pickup the previous ready thread pointer. */
130. 212             ready_previous = thread_ptr -> tx_thread_ready_previous;
131. 213
132. 214             /* 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
133. 215             ready_next -> tx_thread_ready_previous = ready_previous;
134. 216             ready_previous -> tx_thread_ready_next = ready_next;
135. 217
136. 218             /* Determine if this is the head of the priority list. */
137. 219             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)
138. 220         {
139. 221
140. 222             /* Update the head pointer of this priority list. */
141. 223             _tx_thread_priority_list[priority] = ready_next; // 更新就绪线
            程链表表头
142. 224
143. 225 #ifndef TX_DISABLE_PREEMPTION_THRESHOLD
144. 226
145. 227 #if TX_MAX_PRIORITIES > 32
146. 228
147. 229             /* Calculate the index into the bit map array. */
148. 230             map_index = priority/((UINT) 32);
149. 231 #endif
150. 232

```

```

151. 233          /* Check for a thread preempted that had preemption threshold
    set.  */

152. 234          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

153. 235          {

154. 236

155. 237          /* Ensure that this thread's priority is clear in the
    preempt map.  */

156. 238          TX_MOD32_BIT_SET(priority, priority_bit)

157. 239          _tx_thread_preempted_maps[MAP_INDEX] =
    _tx_thread_preempted_maps[MAP_INDEX] & ~(priority_bit)); // 清除抢占标记

158. 240

159. 241 #if TX_MAX_PRIORITIES > 32

160. 242

161. 243          /* Determine if there are any other bits set in this preempt
    map.  */

162. 244          if (_tx_thread_preempted_maps[MAP_INDEX] == ((ULONG) 0))

163. 245          {

164. 246

165. 247          /* No, clear the active bit to signify this preempt map
    has nothing set.  */

166. 248          TX_DIV32_BIT_SET(priority, priority_bit)

167. 249          _tx_thread_preempted_map_active =
    _tx_thread_preempted_map_active & ~(priority_bit);

168. 250          }

169. 251 #endif

170. 252          }

171. 253 #endif

172. 254          }

173. 255          }

174. 256          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)

175. 257          {

176. 258

177. 259          /* This is the only thread at this priority ready to run. Set the
    head

```

```

178. 260                pointer to NULL.  */

179. 261                _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. 262

181. 263 #if TX_MAX_PRIORITIES > 32

182. 264

183. 265                /* Calculate the index into the bit map array.  */

184. 266                map_index = priority/((UINT) 32);

185. 267 #endif

186. 268

187. 269                /* Clear this priority bit in the ready priority bit map.  */

188. 270                TX_MOD32_BIT_SET(priority, priority_bit)

189. 271                _tx_thread_priority_maps[MAP_INDEX] =
                        _tx_thread_priority_maps[MAP_INDEX] & ~(priority_bit)); // 该优先级已经没有任何其他就绪
                        线程了，清除对应的位

190. 272

191. 273 #if TX_MAX_PRIORITIES > 32

192. 274

193. 275                /* Determine if there are any other bits set in this priority map.
                        */

194. 276                if (_tx_thread_priority_maps[MAP_INDEX] == ((ULONG) 0))

195. 277                {

196. 278

197. 279                /* No, clear the active bit to signify this priority map has
                        nothing set.  */

198. 280                TX_DIV32_BIT_SET(priority, priority_bit)

199. 281                _tx_thread_priority_map_active = _tx_thread_priority_map_active
                        & ~(priority_bit));

200. 282                }

201. 283 #endif

202. 284

203. 285 #ifndef TX_DISABLE_PREEMPTION_THRESHOLD

204. 286

205. 287                /* Check for a thread preempted that had preemption-threshold set.
                        */

```

```

206. 288         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)

207. 289         {

208. 290

209. 291             /* Ensure that this thread's priority is clear in the preempt
           map. */

210. 292             TX_MOD32_BIT_SET(priority, priority_bit)

211. 293             _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)

212. 294

213. 295 #if TX_MAX_PRIORITIES > 32

214. 296

215. 297             /* Determine if there are any other bits set in this preempt
           map. */

216. 298             if (_tx_thread_preempted_maps[MAP_INDEX] == ((ULONG) 0))

217. 299             {

218. 300

219. 301                 /* No, clear the active bit to signify this preempted map
           has nothing set. */

220. 302                 TX_DIV32_BIT_SET(priority, priority_bit)

221. 303                 _tx_thread_preempted_map_active =
           _tx_thread_preempted_map_active & ~(priority_bit));

222. 304             }

223. 305 #endif

224. 306         }

225. 307 #endif

226. 308

227. 309 #if TX_MAX_PRIORITIES > 32

228. 310

229. 311             /* Calculate the index to find the next highest priority thread
           ready for execution. */

230. 312             priority_map = _tx_thread_priority_map_active;

231. 313

232. 314             /* Determine if there is anything. */

233. 315             if (priority_map != ((ULONG) 0))

```

```

234. 316      {
235. 317
236. 318          /* Calculate the lowest bit set in the priority map. */
237. 319          TX_LOWEST_SET_BIT_CALCULATE(priority_map, map_index)
238. 320      }
239. 321
240. 322      /* Calculate the base priority as well. */
241. 323      base_priority = map_index * ((UINT) 32);
242. 324 #else
243. 325
244. 326      /* Setup the base priority to zero. */
245. 327      base_priority = ((UINT) 0);
246. 328 #endif
247. 329
248. 330      /* Setup working variable for the priority map. */
249. 331      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)
250. 332
251. 333      /* Make a quick check for no other threads ready for execution. */
252. 334      if (priority_map == ((ULONG) 0)) // 没有就绪线程
253. 335      {
254. 336
255. 337          /* Nothing else is ready. Set highest priority and execute
        thread
256. 338          accordingly. */
257. 339          _tx_thread_highest_priority = ((UINT) TX_MAX_PRIORITIES); //最高
        就绪线程优先级TX_MAX_PRIORITIES
258. 340          _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
259. 341
260. 342 #ifndef TX_MISRA_ENABLE
261. 343
262. 344 #ifdef TX_ENABLE_EVENT_TRACE
263. 345

```

```

264. 346          /* Check that the event time stamp is unchanged.  A different
265. 347          timestamp means that a later event wrote over the thread
266. 348          suspend event. In that case, do nothing here.  */
267. 349          if (entry_ptr != TX_NULL)
268. 350          {
269. 351
270. 352          /* Is the timestamp the same?  */
271. 353          if (time_stamp == entry_ptr ->
tx_trace_buffer_entry_time_stamp)
272. 354          {
273. 355
274. 356          /* Timestamp is the same, set the "next thread pointer"
to the new value of the
275. 357          next thread to execute. This can be used by the trace
analysis tool to keep
276. 358          track of next thread execution.  */
277. 359          entry_ptr -> tx_trace_buffer_entry_information_field_4 =
0;
278. 360          }
279. 361          }
280. 362 #endif
281. 363
282. 364          /* Restore interrupts.  */
283. 365          TX_RESTORE
284. 366
285. 367          /* Determine if preemption should take place. This is only
possible if the current thread pointer is
286. 368          not the same as the execute thread pointer AND the system
state and preempt disable flags are clear.  */
287. 369          TX_THREAD_SYSTEM_RETURN_CHECK(combined_flags) //
_tx_thread_preempt_disable等
288. 370          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)
289. 371          {
290. 372
291. 373 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

```

```

292. 374
293. 375             /* Yes, increment the return to idle return count. */
294. 376             _tx_thread_performance_idle_return_count++;
295. 377 #endif
296. 378
297. 379             /* Preemption is needed - return to the system! */
298. 380             _tx_thread_system_return(); // 返回系统等待就绪线程
299. 381         }
300. 382
301. 383             /* Return to caller. */
302. 384             return;
303. 385 #endif
304. 386     }
305. 387     else // 有就绪线程
306. 388     {
307. 389
308. 390             /* Other threads at different priority levels are ready to run.
309. 391             */
310. 392
311. 393             /* Calculate the lowest bit set in the priority map. */
312. 394             TX_LOWEST_SET_BIT_CALCULATE(priority_map, priority_bit)
313. 395
314. 396             /* Setup the next highest priority variable. */
315. 397             _tx_thread_highest_priority = base_priority + ((UINT)
316. 398             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
317. 399             程)
318. 400         }
319. 401     }
320. 402
321. 403     /* Determine if the suspending thread is the thread designated to
322. 404     execute. */
323. 405
324. 406     if (thread_ptr == _tx_thread_execute_ptr) // 挂起正在执行的线程(没有其他
325. 407     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
326. 408     线程; 挂起不在执行的线程不会导致线程调度)
327. 409
328. 410     {

```



```

321. 403
322. 404          /* Pickup the highest priority thread to execute. */
323. 405          _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
324. 406
325. 407 #ifndef TX_DISABLE_PREEMPTION_THRESHOLD
326. 408
327. 409          /* Determine if a previous thread with preemption-threshold was
          preempted. */
328. 410 #if TX_MAX_PRIORITIES > 32
329. 411          if (_tx_thread_preempted_map_active != ((ULONG) 0))
330. 412 #else
331. 413          if (_tx_thread_preempted_maps[MAP_INDEX] != ((ULONG) 0)) // 检查是否
          有启用抢占的线程被抢占
332. 414 #endif
333. 415          {
334. 416
335. 417          /* Yes, there was a thread preempted when it was using
          preemption-threshold. */
336. 418
337. 419          /* Disable preemption. */
338. 420          _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)
339. 421
340. 422          /* Restore interrupts. */
341. 423          TX_RESTORE
342. 424
343. 425          /* Interrupts are enabled briefly here to keep the interrupt
          lockout time deterministic. */
344. 426
345. 427
346. 428          /* Disable interrupts again. */
347. 429          TX_DISABLE
348. 430
349. 431          /* Decrement the preemption disable variable. */

```

```

350. 432         _tx_thread_preempt_disable--;
351. 433
352. 434         /* Calculate the thread with preemption threshold set that
353. 435             was interrupted by a thread above the preemption level. */
354. 436
355. 437 #if TX_MAX_PRIORITIES > 32
356. 438
357. 439         /* Calculate the index to find the next highest priority thread
358. 440            ready for execution. */
359. 441
360. 442         /* Calculate the lowest bit set in the priority map. */
361. 443         TX_LOWEST_SET_BIT_CALCULATE(priority_map, map_index)
362. 444
363. 445         /* Calculate the base priority as well. */
364. 446         base_priority = map_index * ((UINT) 32);
365. 447 #else
366. 448
367. 449         /* Setup the base priority to zero. */
368. 450         base_priority = ((UINT) 0);
369. 451 #endif
370. 452
371. 453         /* Setup temporary preempted map. */
372. 454         priority_map = _tx_thread_preempted_maps[MAP_INDEX];
373. 455
374. 456         /* Calculate the lowest bit set in the priority map. */
375. 457         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)
376. 458
377. 459         /* Setup the highest priority preempted thread. */
378. 460         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

```

```

379. 461
380. 462             /* Determine if the next highest priority thread is above the
             highest priority threshold value.  */
381. 463             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
382. 464             {
383. 465
384. 466             /* Thread not allowed to execute until earlier preempted
                thread finishes or lowers its
385. 467                 preemption-threshold.  */
386. 468                 _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)
387. 469
388. 470             /* Clear the corresponding bit in the preempted map, since
                the preemption has been restored.  */
389. 471                 TX_MOD32_BIT_SET(priority, priority_bit)
390. 472                 _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
391. 473
392. 474 #if TX_MAX_PRIORITIES > 32
393. 475
394. 476             /* Determine if there are any other bits set in this preempt
                map.  */
395. 477                 if (_tx_thread_preempted_maps[MAP_INDEX] == ((ULONG) 0))
396. 478                 {
397. 479
398. 480                 /* No, clear the active bit to signify this preempt map
                has nothing set.  */
399. 481                 TX_DIV32_BIT_SET(priority, priority_bit)
400. 482                 _tx_thread_preempted_map_active =
                _tx_thread_preempted_map_active & ~(priority_bit));
401. 483             }
402. 484 #endif
403. 485         }

```

```

404. 486      }

405. 487 #endif

406. 488

407. 489 #ifndef TX_MISRA_ENABLE

408. 490

409. 491 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

410. 492

411. 493      /* Is the execute pointer different? */

412. 494      if
      (_tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] !=
      _tx_thread_execute_ptr)

413. 495      {

414. 496

415. 497          /* Move to next entry. */

416. 498          _tx_thread_performance__execute_log_index++;

417. 499

418. 500          /* Check for wrap condition. */

419. 501          if (_tx_thread_performance__execute_log_index >=
      TX_THREAD_EXECUTE_LOG_SIZE)

420. 502          {

421. 503

422. 504              /* Set the index to the beginning. */

423. 505              _tx_thread_performance__execute_log_index = ((UINT) 0);

424. 506          }

425. 507

426. 508          /* Log the new execute pointer. */

427. 509          _tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] =
      _tx_thread_execute_ptr;

428. 510      }

429. 511 #endif

430. 512

431. 513 #ifdef TX_ENABLE_EVENT_TRACE

432. 514

```

```

433. 515          /* Check that the event time stamp is unchanged.  A different
434. 516          timestamp means that a later event wrote over the thread
435. 517          suspend event. In that case, do nothing here.  */
436. 518          if (entry_ptr != TX_NULL)
437. 519          {
438. 520
439. 521          /* Is the timestamp the same?  */
440. 522          if (time_stamp == entry_ptr -> tx_trace_buffer_entry_time_stamp)
441. 523          {
442. 524
443. 525          /* Timestamp is the same, set the "next thread pointer" to
the new value of the
444. 526          next thread to execute. This can be used by the trace
analysis tool to keep
445. 527          track of next thread execution.  */
446. 528          entry_ptr -> tx_trace_buffer_entry_information_field_4 =
TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
447. 529          }
448. 530          }
449. 531 #endif
450. 532
451. 533          /* Restore interrupts.  */
452. 534          TX_RESTORE
453. 535
454. 536          /* Determine if preemption should take place. This is only possible
if the current thread pointer is
455. 537          not the same as the execute thread pointer AND the system state
and preempt disable flags are clear.  */
456. 538          TX_THREAD_SYSTEM_RETURN_CHECK(combined_flags)
457. 539          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
458. 540          {
459. 541
460. 542 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
461. 543

```

```

462. 544                /* No, there is another thread ready to run and will be
    scheduled upon return.  */

463. 545                _tx_thread_performance_non_idle_return_count++;

464. 546 #endif

465. 547

466. 548                /* Preemption is needed - return to the system!  */

467. 549                _tx_thread_system_return();

468. 550        }

469. 551

470. 552        /* Return to caller.  */

471. 553        return; // 返回

472. 554 #endif

473. 555     }

474. 556

475. 557 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

476. 558

477. 559        /* Is the execute pointer different?  */

478. 560        if
    (_tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] !=
    _tx_thread_execute_ptr)

479. 561        {

480. 562

481. 563                /* Move to next entry.  */

482. 564                _tx_thread_performance__execute_log_index++;

483. 565

484. 566                /* Check for wrap condition.  */

485. 567                if (_tx_thread_performance__execute_log_index >=
    TX_THREAD_EXECUTE_LOG_SIZE)

486. 568                {

487. 569

488. 570                /* Set the index to the beginning.  */

489. 571                _tx_thread_performance__execute_log_index = ((UINT) 0);

490. 572                }

491. 573

```

```

492. 574          /* Log the new execute pointer.  */
493. 575
    _tx_thread_performance_execute_log[_tx_thread_performance__execute_log_index] =
        _tx_thread_execute_ptr;
494. 576      }
495. 577 #endif
496. 578
497. 579 #ifdef TX_ENABLE_EVENT_TRACE
498. 580
499. 581          /* Check that the event time stamp is unchanged.  A different
500. 582          timestamp means that a later event wrote over the thread
501. 583          suspend event. In that case, do nothing here.  */
502. 584          if (entry_ptr != TX_NULL)
503. 585          {
504. 586
505. 587              /* Is the timestamp the same?  */
506. 588              if (time_stamp == entry_ptr -> tx_trace_buffer_entry_time_stamp)
507. 589              {
508. 590
509. 591                  /* Timestamp is the same, set the "next thread pointer" to the
                    new value of the
510. 592                  next thread to execute. This can be used by the trace
                    analysis tool to keep
511. 593                  track of next thread execution.  */
512. 594 #ifdef TX_MISRA_ENABLE
513. 595                  entry_ptr -> tx_trace_buffer_entry_info_4 =
                    TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
514. 596 #else
515. 597                  entry_ptr -> tx_trace_buffer_entry_information_field_4 =
                    TX_POINTER_TO_ULONG_CONVERT(_tx_thread_execute_ptr);
516. 598 #endif
517. 599              }
518. 600          }
519. 601 #endif
520. 602      }

```

```

521. 603
522. 604     /* Restore interrupts. */
523. 605     TX_RESTORE
524. 606
525. 607     /* Determine if a preemption condition is present. */
526. 608     if (current_thread != _tx_thread_execute_ptr)
527. 609     {
528. 610
529. 611 #ifdef TX_ENABLE_STACK_CHECKING
530. 612
531. 613         /* Pickup the next execute pointer. */
532. 614         thread_ptr = _tx_thread_execute_ptr;
533. 615
534. 616         /* Check this thread's stack. */
535. 617         TX_THREAD_STACK_CHECK(thread_ptr)
536. 618 #endif
537. 619
538. 620         /* Determine if preemption should take place. This is only possible if
           the current thread pointer is
539. 621         not the same as the execute thread pointer AND the system state and
           preempt disable flags are clear. */
540. 622         TX_THREAD_SYSTEM_RETURN_CHECK(combined_flags)
541. 623         if (combined_flags == ((ULONG) 0))
542. 624         {
543. 625
544. 626 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
545. 627
546. 628             /* Determine if an idle system return is present. */
547. 629             if (_tx_thread_execute_ptr == TX_NULL)
548. 630             {
549. 631
550. 632                 /* Yes, increment the return to idle return count. */
551. 633                 _tx_thread_performance_idle_return_count++;

```



```
552. 634      }
553. 635      else
554. 636      {
555. 637
556. 638          /* No, there is another thread ready to run and will be
           scheduled upon return. */
557. 639          _tx_thread_performance_non_idle_return_count++;
558. 640      }
559. 641 #endif
560. 642
561. 643      /* Preemption is needed - return to the system! */
562. 644      _tx_thread_system_return();
563. 645  }
564. 646  }
565. 647
566. 648  /* Return to caller. */
567. 649  return;
568. 650 }
```

