(163条消息) ThreadX内核源码分析 - 线程同步之互斥锁及动态优先级_arm7star的博客-CSDN博客_互斥锁可能导致线程 未执行

1、ThreadX互斥锁介绍

互斥锁一般用来锁多线程都需要访问的临界资源，这些资源不能并发操作，例如对某个内存的互斥读写，这个读写很快，但是不能多线程同时进行，所以需要加互斥锁；

占用互斥锁的线程没能执行，可能是被高优先级线程抢占了或者临界资源里面调用了阻塞操作；高优先级线程想要访问临界资源就必须等待低优先级线程被调度然后释放互斥锁，如果低优先级线程不提高优先级不被执行的话，那么高优先级的线程就会阻塞，直到所有高优先级线程都阻塞了，才能轮到占用互斥锁的低优先级线程执行，这样高优先级线程就得不到及时处理，通常高优先级线程的任务都比较紧急；

ThreadX互斥锁实现了优先级继承，如果等待互斥锁的线程的优先级高于占用互斥锁的线程的优先级，那么将占用互斥锁线程的优先级调整到与等待互斥锁线程的优先级一样高，然占用互斥锁的线程尽快执行，尽快释放互斥锁，释放互斥锁时，线程优先级恢复原始优先级，让出cpu执行；

另外ThreadX互斥锁是可重复获取的，获取到了互斥锁的线程可以多次获取互斥锁，0表示互斥锁没有占用，非0表示互斥锁被占用，每获取一次互斥锁，互斥锁计数加1，获取互斥锁次数要与释放互斥锁次数相同，否则互斥锁不会被释放。

2、互斥锁获取_tx_mutex_get

2.1、互斥锁的获取_tx_mutex_get

互斥锁获取主要看互斥锁计数器：

- 如果互斥锁计数器为0，那么互斥锁没有被占用，获取互斥锁，互斥锁继承当前线程的优先级，互斥锁计数器加1
- 如果互斥锁已经被当前线程占用，互斥锁计数器加1即可；
- 如果互斥锁被其他线程占用，当前线程不能阻塞或者没有设置等待参数就返回互斥锁不可用，否则挂起当前线程，当前线程加入等待互斥锁线程链表，更新tx_mutex_highest_priority_waiting(所有等待互斥锁的线程的最高优先级)，如果当前等待互斥锁线程的优先级高于占用互斥锁线程的优先级，那么将占用互斥锁线程的优先级设置改变成当前线程的优先级，让占用互斥锁的线程尽快处理临界资源并释放互斥锁，等待互斥锁的高优先级线程才能尽快获取到互斥锁。

_tx_mutex_get代码实现如下：

```
1. 077 UINT  _tx_mutex_get(TX_MUTEX *mutex_ptr, ULONG wait_option)
2. 078 {
3. 079
4. 080 TX_INTERRUPT_SAVE_AREA
5. 081
6. 082 TX_THREAD       *thread_ptr;
7. 083 TX_MUTEX        *next_mutex;
8. 084 TX_MUTEX        *previous_mutex;
9. 085 TX_THREAD       *mutex_owner;
10. 086 TX_THREAD       *next_thread;
11. 087 TX_THREAD       *previous_thread;
12. 088 UINT            status;
13. 089
14. 090
15. 091     /* Disable interrupts to get an instance from the mutex.  */
16. 092     TX_DISABLE
17. 093
18. 094 #ifdef TX_MUTEX_ENABLE_PERFORMANCE_INFO
19. 095
20. 096     /* Increment the total mutex get counter.  */
21. 097     _tx_mutex_performance_get_count++;
22. 098
23. 099     /* Increment the number of attempts to get this mutex.  */
24. 100     mutex_ptr -> tx_mutex_performance_get_count++;
25. 101 #endif
26. 102
27. 103     /* If trace is enabled, insert this event into the trace buffer.  */
28. 104     TX_TRACE_IN_LINE_INSERT(TX_TRACE_MUTEX_GET, mutex_ptr, wait_option,
    TX_POINTER_TO_ULONG_CONVERT(mutex_ptr -> tx_mutex_owner), mutex_ptr ->
    tx_mutex_ownership_count, TX_TRACE_MUTEX_EVENTS)
29. 105
30. 106     /* Log this kernel call.  */
```

```
107     TX_EL_MUTEX_GET_INSERT
108
109     /* Pickup thread pointer.  */
110     TX_THREAD_GET_CURRENT(thread_ptr)
111
112     /* Determine if this mutex is available.  */
113     if (mutex_ptr -> tx_mutex_ownership_count == ((UINT) 0)) // 互斥锁没有被占用
114     {
115
116         /* Set the ownership count to 1.  */
117         mutex_ptr -> tx_mutex_ownership_count =  ((UINT) 1); // 互斥锁计数加1(互斥锁被占用)
118
119         /* Remember that the calling thread owns the mutex.  */
120         mutex_ptr -> tx_mutex_owner =  thread_ptr; // 记录占用互斥锁的线程为当前线程
121
122         /* Determine if the thread pointer is valid.  */
123         if (thread_ptr != TX_NULL) // 当前线程是否有效(非线程上下文调用不用执行if分支，正常情况下都是在线程上下文调用互斥锁)
124         {
125
126             /* Determine if priority inheritance is required.  */
127             if (mutex_ptr -> tx_mutex_inherit == TX_TRUE)
128             {
129
130                 /* Remember the current priority of thread.  */
131                 mutex_ptr -> tx_mutex_original_priority =   thread_ptr -> tx_thread_priority; // tx_mutex_original_priority记录线程获取互斥锁时的原始优先级(第一次获取互斥锁的时候才会执行到这里)
132
133                 /* Setup the highest priority waiting thread.  */
134                 mutex_ptr -> tx_mutex_highest_priority_waiting =  ((UINT) TX_MAX_PRIORITIES); // 正在等待互斥锁的线程的最高优先级，没有时默认就是TX_MAX_PRIORITIES(最低优先级)
```

```
135             }
136
137             /* Pickup next mutex pointer, which is the head of the list.  */
138             next_mutex =  thread_ptr -> tx_thread_owned_mutex_list; // 线程占用互斥锁链表
139
140             /* Determine if this thread owns any other mutexes that have priority inheritance.  */
141             if (next_mutex != TX_NULL) // 线程占用其他互斥锁，将当前获取的互斥锁加入链表表头
142             {
143
144                 /* Non-empty list. Link up the mutex.  */
145
146                 /* Pickup the next and previous mutex pointer.  */
147                 previous_mutex =  next_mutex -> tx_mutex_owned_previous;
148
149                 /* Place the owned mutex in the list.  */
150                 next_mutex -> tx_mutex_owned_previous =  mutex_ptr;
151                 previous_mutex -> tx_mutex_owned_next =  mutex_ptr;
152
153                 /* Setup this mutex's next and previous created links.  */
154                 mutex_ptr -> tx_mutex_owned_previous =  previous_mutex;
155                 mutex_ptr -> tx_mutex_owned_next =      next_mutex;
156             }
157             else // 线程没有占用其他互斥锁，新建一个互斥锁链表，只有一个当前获取到的互斥锁
158             {
159
160                 /* The owned mutex list is empty.  Add mutex to empty list.  */
161                 thread_ptr -> tx_thread_owned_mutex_list =     mutex_ptr;
162                 mutex_ptr -> tx_mutex_owned_next =             mutex_ptr;
163                 mutex_ptr -> tx_mutex_owned_previous =         mutex_ptr;
164             }
```

```
165
166             /* Increment the number of mutexes owned counter.  */
167             thread_ptr -> tx_thread_owned_mutex_count++; // 线程占用互斥锁个数加1
168         }
169
170         /* Restore interrupts.  */
171         TX_RESTORE
172
173         /* Return success.  */
174         status =  TX_SUCCESS;
175     }
176
177     /* Otherwise, see if the owning thread is trying to obtain the same mutex.  */
178     else if (mutex_ptr -> tx_mutex_owner == thread_ptr) // 当前线程之前就已经占用了该互斥锁，再次获取该互斥锁，只需要增加互斥锁计数即可
179     {
180
181         /* The owning thread is requesting the mutex again, just
182            increment the ownership count.  */
183         mutex_ptr -> tx_mutex_ownership_count++; // 互斥锁占用次数加1
184
185         /* Restore interrupts.  */
186         TX_RESTORE
187
188         /* Return success.  */
189         status =  TX_SUCCESS;
190     }
191     else // 互斥锁被其他线程占用
192     {
193
194         /* Determine if the request specifies suspension.  */
```

```
119. 195         if (wait_option != TX_NO_WAIT) // 有设置等待选项
120. 196         {
121. 197
122. 198             /* Determine if the preempt disable flag is non-zero.  */
123. 199             if (_tx_thread_preempt_disable != ((UINT) 0)) // 如果线程禁止抢占的
     话，那么不能阻塞当前线程，返回互斥锁不可用即可，否则其他线程被禁止抢占了就得不到执行
124. 200             {
125. 201
126. 202                 /* Restore interrupts.  */
127. 203                 TX_RESTORE
128. 204
129. 205                 /* Suspension is not allowed if the preempt disable flag is non-
     zero at this point - return error completion.  */
130. 206                 status =  TX_NOT_AVAILABLE;
131. 207             }
132. 208             else // 线程没有禁止抢占，那么需要挂起当前线程
133. 209             {
134. 210
135. 211                 /* Prepare for suspension of this thread.  */
136. 212
137. 213                 /* Pickup the mutex owner.  */
138. 214                 mutex_owner =  mutex_ptr -> tx_mutex_owner; // 获取占用互斥锁的
     线程
139. 215
140. 216 #ifdef TX_MUTEX_ENABLE_PERFORMANCE_INFO
141. 217
142. 218                 /* Increment the total mutex suspension counter.  */
143. 219                 _tx_mutex_performance_suspension_count++;
144. 220
145. 221                 /* Increment the number of suspensions on this mutex.  */
146. 222                 mutex_ptr -> tx_mutex_performance_suspension_count++;
147. 223
148. 224                 /* Determine if a priority inversion is present.  */
```

```
149. 225                if (thread_ptr -> tx_thread_priority < mutex_owner ->
     tx_thread_priority)

150. 226                {

151. 227

152. 228                    /* Yes, priority inversion is present!  */

153. 229

154. 230                    /* Increment the total mutex priority inversions counter.
     */

155. 231                    _tx_mutex_performance_priority_inversion_count++;

156. 232

157. 233                    /* Increment the number of priority inversions on this
     mutex.  */

158. 234                    mutex_ptr ->
     tx_mutex_performance_priority_inversion_count++;

159. 235

160. 236 #ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO

161. 237

162. 238                    /* Increment the number of total thread priority inversions.
     */

163. 239                    _tx_thread_performance_priority_inversion_count++;

164. 240

165. 241                    /* Increment the number of priority inversions for this
     thread.  */

166. 242                    thread_ptr ->
     tx_thread_performance_priority_inversion_count++;

167. 243 #endif

168. 244                }

169. 245 #endif

170. 246

171. 247                /* Setup cleanup routine pointer.  */

172. 248                thread_ptr -> tx_thread_suspend_cleanup =  &(_tx_mutex_cleanup);
     // 等待互斥锁超时或者被中断时的清理函数(等待超时需要通过定时调用_tx_mutex_cleanup唤
     醒阻塞的线程)

173. 249

174. 250                /* Setup cleanup information, i.e. this mutex control

175. 251                   block.  */
```

```
176. 252                thread_ptr -> tx_thread_suspend_control_block =  (VOID *)
     mutex_ptr; // 清理函数的参数(等待的互斥锁)，线程会挂到互斥锁等待链表里面，超时时需要
     从等待链表里面删除当前线程

177. 253

178. 254 #ifndef TX_NOT_INTERRUPTABLE

179. 255

180. 256                /* Increment the suspension sequence number, which is used to
     identify

181. 257                   this suspension event.  */

182. 258                thread_ptr -> tx_thread_suspension_sequence++;

183. 259 #endif

184. 260

185. 261                /* Setup suspension list.  */

186. 262                if (mutex_ptr -> tx_mutex_suspended_count == TX_NO_SUSPENSIONS)
     // 当前线程加入等待mutex_ptr的挂起线程链表tx_mutex_suspension_list里面

187. 263                {

188. 264

189. 265                    /* No other threads are suspended.  Setup the head pointer
     and

190. 266                       just setup this threads pointers to itself.  */

191. 267                    mutex_ptr -> tx_mutex_suspension_list =         thread_ptr;

192. 268                    thread_ptr -> tx_thread_suspended_next =        thread_ptr;

193. 269                    thread_ptr -> tx_thread_suspended_previous =    thread_ptr;

194. 270                }

195. 271                else

196. 272                {

197. 273

198. 274                    /* This list is not NULL, add current thread to the end. */

199. 275                    next_thread =                                   mutex_ptr ->
     tx_mutex_suspension_list;

200. 276                    thread_ptr -> tx_thread_suspended_next =        next_thread;

201. 277                    previous_thread =                               next_thread
     -> tx_thread_suspended_previous;

202. 278                    thread_ptr -> tx_thread_suspended_previous =
     previous_thread;

203. 279                    previous_thread -> tx_thread_suspended_next =   thread_ptr;
```

```
204. 280                    next_thread -> tx_thread_suspended_previous =   thread_ptr;

205. 281                }

206. 282

207. 283                /* Increment the suspension count.  */

208. 284                mutex_ptr -> tx_mutex_suspended_count++; // 等待互斥锁的线程计数
     加1

209. 285

210. 286                /* Set the state to suspended.  */

211. 287                thread_ptr -> tx_thread_state =    TX_MUTEX_SUSP; // 当前线程修
     改为挂起状态

212. 288

213. 289 #ifdef TX_NOT_INTERRUPTABLE

214. 290

215. 291                /* Determine if we need to raise the priority of the thread

216. 292                   owning the mutex.  */

217. 293                if (mutex_ptr -> tx_mutex_inherit == TX_TRUE)

218. 294                {

219. 295

220. 296                    /* Determine if this is the highest priority to raise for
     this mutex.  */

221. 297                    if (mutex_ptr -> tx_mutex_highest_priority_waiting >
     thread_ptr -> tx_thread_priority)

222. 298                    {

223. 299

224. 300                        /* Remember this priority.  */

225. 301                        mutex_ptr -> tx_mutex_highest_priority_waiting =
     thread_ptr -> tx_thread_priority;

226. 302                    }

227. 303

228. 304                    /* Determine if we have to update inherit priority level of
     the mutex owner.  */

229. 305                    if (thread_ptr -> tx_thread_priority < mutex_owner ->
     tx_thread_inherit_priority)

230. 306                    {

231. 307
```

```
232. 308                        /* Remember the new priority inheritance priority.  */
233. 309                        mutex_owner -> tx_thread_inherit_priority =  thread_ptr
     -> tx_thread_priority;
234. 310                    }
235. 311
236. 312                    /* Priority inheritance is requested, check to see if the
     thread that owns the mutex is lower priority.  */
237. 313                    if (mutex_owner -> tx_thread_priority > thread_ptr ->
     tx_thread_priority)
238. 314                    {
239. 315
240. 316                        /* Yes, raise the suspended, owning thread's priority to
     that
241. 317                           of the current thread.  */
242. 318                        _tx_mutex_priority_change(mutex_owner, thread_ptr ->
     tx_thread_priority);
243. 319
244. 320 #ifdef TX_MUTEX_ENABLE_PERFORMANCE_INFO
245. 321
246. 322                        /* Increment the total mutex priority inheritance
     counter.  */
247. 323                        _tx_mutex_performance__priority_inheritance_count++;
248. 324
249. 325                        /* Increment the number of priority inheritance
     situations on this mutex.  */
250. 326                        mutex_ptr ->
     tx_mutex_performance__priority_inheritance_count++;
251. 327 #endif
252. 328                    }
253. 329                }
254. 330
255. 331                /* Call actual non-interruptable thread suspension routine.  */
256. 332                _tx_thread_system_ni_suspend(thread_ptr, wait_option);
257. 333
258. 334                /* Restore interrupts.  */
259. 335                TX_RESTORE
```

```
336 #else

337

338                 /* Set the suspending flag.  */

339                 thread_ptr -> tx_thread_suspending =  TX_TRUE; // 设置线程正在挂起(线程还没从就绪链表删除，还没真正挂起，其他挂起或者唤醒当前线程的操作需要检查tx_thread_suspending，不能唤醒挂起中的线程，延迟挂起挂起中线程，也就是互斥锁挂起之后，下次唤醒线程时再执行之前的挂起操作)

340

341                 /* Setup the timeout period.  */

342                 thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks =  wait_option; // 线程超时时间(挂起线程时会启动一个定时器，定时器超时调用前面设置的回调函数，唤醒阻塞线程)

343

344                 /* Temporarily disable preemption.  */

345                 _tx_thread_preempt_disable++; // 禁止抢占，后面会打开中断，避免当前线程被抢占切换出去，等线程真正挂起之后才调度其他线程，线程挂起之后，后面还有部分重要操作要执行

346

347                 /* Restore interrupts.  */

348                 TX_RESTORE

349

350                 /* Determine if we need to raise the priority of the thread

351                    owning the mutex.  */

352                 if (mutex_ptr -> tx_mutex_inherit == TX_TRUE) // 如果互斥锁继承为TX_TRUE，那么占用互斥锁的线程可以继承等待互斥锁线程的优先级

353                 {

354

355                     /* Determine if this is the highest priority to raise for this mutex.  */

356                     if (mutex_ptr -> tx_mutex_highest_priority_waiting > thread_ptr -> tx_thread_priority) // 当前线程是所有等待互斥锁的线程里面优先级最高的线程

357                     {

358

359                         /* Remember this priority.  */

360                         mutex_ptr -> tx_mutex_highest_priority_waiting = thread_ptr -> tx_thread_priority; // 更新等待互斥锁线程的最高优先级

361                     }
```

```
286. 362

287. 363                    /* Determine if we have to update inherit priority level of
     the mutex owner.  */

288. 364                    if (thread_ptr -> tx_thread_priority < mutex_owner ->
     tx_thread_inherit_priority) // 当前线程的优先级高于占用互斥锁线程的继承优先级(线程的
     继承优先级默认是TX_MAX_PRIORITIES)

289. 365                    {

290. 366

291. 367                        /* Remember the new priority inheritance priority.  */

292. 368                        mutex_owner -> tx_thread_inherit_priority =  thread_ptr
     -> tx_thread_priority; // 更新占用互斥锁线程的继承优先级

293. 369                    }

294. 370

295. 371                    /* Priority inheritance is requested, check to see if the
     thread that owns the mutex is lower priority.  */

296. 372                    if (mutex_owner -> tx_thread_priority > thread_ptr ->
     tx_thread_priority) // 占用互斥的锁线程的优先级低于当前等待互斥锁线程的优先级

297. 373                    {

298. 374

299. 375                        /* Yes, raise the suspended, owning thread's priority to
     that

300. 376                           of the current thread.  */

301. 377                        _tx_mutex_priority_change(mutex_owner, thread_ptr ->
     tx_thread_priority); // 调整占用互斥锁线程的优先级为当前线程的优先级(让占用互斥锁的
     线程尽快执行，然后释放互斥锁)

302. 378

303. 379 #ifdef TX_MUTEX_ENABLE_PERFORMANCE_INFO

304. 380

305. 381                        /* Increment the total mutex priority inheritance
     counter.  */

306. 382                        _tx_mutex_performance__priority_inheritance_count++;

307. 383

308. 384                        /* Increment the number of priority inheritance
     situations on this mutex.  */

309. 385                        mutex_ptr ->
     tx_mutex_performance__priority_inheritance_count++;

310. 386 #endif

311. 387                    }
```

```
388                 }
389
390                 /* Call actual thread suspension routine.  */
391                 _tx_thread_system_suspend(thread_ptr); // 挂起当前线程
392 #endif
393                 /* Return the completion status.  */
394                 status =  thread_ptr -> tx_thread_suspend_status;
395             }
396         }
397         else // 没有等待选项，不等待互斥锁，那么返回互斥锁不可用即可
398         {
399
400             /* Restore interrupts.  */
401             TX_RESTORE
402
403             /* Immediate return, return error completion.  */
404             status =  TX_NOT_AVAILABLE;
405         }
406     }
407
408     /* Return completion status.  */
409     return(status);
410 }
```

2.2、线程优先级调整_tx_mutex_priority_change

调整优先级主要是将被改变优先级的线程从原来的就绪线程链表删除，加入到新的就绪线程链表，这个过程就导致被改变优先级的线程被加入到就绪线程链表末尾，而且还会导致内存重新选择下一个执行线程，例如：当前线程优先级为1，优先级1的线程只要当前线程就绪，优先级2有就绪线程，当前线程把自己的优先级将为2了，内核选择原来优先级2就绪链表的第一个就绪线程作为下一个执行的线程，很明显，应该让当前线程继续执行，那么就得再次把当前线程移动到表头，再例如：当前线程优先级为1，优先级1的线程只要当前线程就绪，就绪线程的次高优先级为5，优先级5的第一个就绪线程的抢占阈值为3，当前线程需要降低优先级为3，当前线程挂起时，内核选择优先级5的线程作为下一个执行线程，唤醒当前线程，虽然当前线程是最高优先级，但是优先级5的线程的抢占阈值为3，当前线程不能抢占阈值3的线程，理论上应该是抢占阈值3的线程不能抢占当前线程，当前线

程就因为改变优先级而被不能抢占的线程抢占了，那么还得抢回来，将当前线程设置为下一个执行线程，并且移动到表头。(有些场景没看到内核怎么确保改变优先级的线程仍然在表头，而且有场景已经确认不能保证被标记抢占的线程不在就绪线程链表表头，不明白是**bug**能还是内核设计如此，向社区提交了**issue**，等待社区回复再具体分析**!!!**)

_tx_mutex_priority_change实现代码如下：

```
 1. 083 VOID  _tx_mutex_priority_change(TX_THREAD *thread_ptr, UINT new_priority)
 2. 084 {
 3. 085
 4. 086 #ifndef TX_NOT_INTERRUPTABLE
 5. 087
 6. 088 TX_INTERRUPT_SAVE_AREA
 7. 089 #endif
 8. 090
 9. 091 TX_THREAD       *execute_ptr;
10. 092 TX_THREAD       *next_execute_ptr;
11. 093 UINT            original_priority;
12. 094 #ifndef TX_DISABLE_PREEMPTION_THRESHOLD
13. 095 ULONG           priority_bit;
14. 096 #if TX_MAX_PRIORITIES > 32
15. 097 UINT            map_index;
16. 098 #endif
17. 099 #endif
18. 100
19. 101
20. 102
21. 103 #ifndef TX_NOT_INTERRUPTABLE
22. 104
23. 105     /* Lockout interrupts while the thread is being suspended.  */
24. 106     TX_DISABLE
25. 107 #endif
26. 108
27. 109     /* Determine if this thread is currently ready.  */
28. 110     if (thread_ptr -> tx_thread_state != TX_READY) // 如果线程非就绪状态，简单改
    变线程的优先级、抢占阈值，因为阻塞线程不在就绪链表里面，不需要移动(抢占阈值不低于线
    程创建时用户指定的抢占阈值；线程占用互斥锁时，线程会继承高优先级等待互斥锁线程的优先
    级，互斥锁释放时，会降低恢复旧的优先级)
29. 111     {
30. 112
```

```
113         /* Change thread priority to the new mutex priority-inheritance
    priority.  */
114         thread_ptr -> tx_thread_priority =  new_priority; // 设置线程的新的优先
    级
115
116         /* Determine how to setup the thread's preemption-threshold.  */
117         if (thread_ptr -> tx_thread_user_preempt_threshold < new_priority)
118         {
119
120             /* Change thread preemption-threshold to the user's preemption-
    threshold.  */
121             thread_ptr -> tx_thread_preempt_threshold =  thread_ptr ->
    tx_thread_user_preempt_threshold;
122         }
123         else
124         {
125
126             /* Change the thread preemption-threshold to the new threshold.  */
127             thread_ptr -> tx_thread_preempt_threshold =  new_priority;
128         }
129
130 #ifndef TX_NOT_INTERRUPTABLE
131         /* Restore interrupts.  */
132         TX_RESTORE
133 #endif
134     }
135     else // 就绪状态的线程，改变优先级后，线程需要移动到新的优先级的就绪线程链表
    里面(从原来链表删除，再添加到新的就绪线程链表)
136     {
137
138         /* Pickup the next thread to execute.  */
139         execute_ptr =  _tx_thread_execute_ptr; // 保存当前时间下一个需要执行的线
    程
140
141         /* Save the original priority.  */
```

```
142         original_priority =  thread_ptr -> tx_thread_priority; // 记录线程的原始优先级(改变优先级前的优先级)
143
144 #ifdef TX_NOT_INTERRUPTABLE
145
146         /* Increment the preempt disable flag.  */
147         _tx_thread_preempt_disable++;
148
149         /* Set the state to priority change.  */
150         thread_ptr -> tx_thread_state =    TX_PRIORITY_CHANGE;
151
152         /* Call actual non-interruptable thread suspension routine.  */
153         _tx_thread_system_ni_suspend(thread_ptr, ((ULONG) 0));
154
155         /* At this point, the preempt disable flag is still set, so we still have
156            protection against all preemption.  */
157
158         /* Change thread priority to the new mutex priority-inheritance priority.  */
159         thread_ptr -> tx_thread_priority =  new_priority;
160
161         /* Determine how to setup the thread's preemption-threshold.  */
162         if (thread_ptr -> tx_thread_user_preempt_threshold < new_priority)
163         {
164
165             /* Change thread preemption-threshold to the user's preemption-threshold.  */
166             thread_ptr -> tx_thread_preempt_threshold =  thread_ptr -> tx_thread_user_preempt_threshold;
167         }
168         else
169         {
170
```

```
 89. 171             /* Change the thread preemption-threshold to the new threshold.  */
 90. 172             thread_ptr -> tx_thread_preempt_threshold =  new_priority;
 91. 173         }
 92. 174
 93. 175         /* Resume the thread with the new priority.  */
 94. 176         _tx_thread_system_ni_resume(thread_ptr);
 95. 177
 96. 178         /* Decrement the preempt disable flag.  */
 97. 179         _tx_thread_preempt_disable--;
 98. 180 #else
 99. 181
100. 182         /* Increment the preempt disable flag.  */
101. 183         _tx_thread_preempt_disable =  _tx_thread_preempt_disable + ((UINT) 2);
     // 禁止抢占计数器加2(主要因为移动线程到其他就绪线程链表需要挂起再唤醒该线程，挂起/唤醒会对禁止抢占计数器减1，如果_tx_thread_preempt_disable减为0，就会进行调度，对抢占计数器加2就是为了在挂起/唤醒过程中，禁止抢占当前线程，不管是否有高优先级线程，当前线程需要继续处理后续代码；调用_tx_mutex_priority_change函数前的代码已经禁止抢占了，所以_tx_mutex_priority_change不会被其他线程抢占!!!)
102. 184
103. 185         /* Set the state to priority change.  */
104. 186         thread_ptr -> tx_thread_state =    TX_PRIORITY_CHANGE; // 将线程状态设置为TX_PRIORITY_CHANGE(主要是在挂起操作时，对TX_PRIORITY_CHANGE以及其他特殊状态的线程进行挂起，并不会立即挂起，而是会设置一个延迟挂起的标志，这些状态正在处理比较重要的事情，这个过程不能被挂起；延迟挂起被设置，那么下次唤醒线程时，就会执行延迟的挂起操作，挂起线程)
105. 187
106. 188         /* Set the suspending flag. */
107. 189         thread_ptr -> tx_thread_suspending =  TX_TRUE; // 设置挂起中(挂起中表示线程还没从就绪线程链表删除，如果有其他操作唤醒挂起中的线程，那么将状态改成就绪即可，因为线程还在就绪线程链表，不需要再次添加到就绪线程链表；_tx_thread_system_suspend只会对挂起中的线程进行挂起操作，线程的挂起中如果不为真，那么，可能线程挂起过程中，被其他线程唤醒了，就不要再继续挂起线程)
108. 190
109. 191         /* Setup the timeout period.  */
110. 192         thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks =
     ((ULONG) 0); // 本次挂起不需要启动定时器(本次挂起的目的主要是将线程从就绪线程链表删除)
111. 193
112. 194         /* Restore interrupts.  */
```

```
113. 195         TX_RESTORE
114. 196
115. 197         /* The thread is ready and must first be removed from the list.  Call
     the
116. 198            system suspend function to accomplish this.  */
117. 199         _tx_thread_system_suspend(thread_ptr); // 挂起线程，从就绪线程链表删除线
     程，会对禁止抢占计数器减1(_tx_thread_system_suspend并不知道本次挂起的目的，挂起
     thread_ptr，可能会选一个新的下一个被执行的线程，所以前面先记录了下一个被执行的线程)
118. 200
119. 201         /* Disable interrupts.  */
120. 202         TX_DISABLE
121. 203
122. 204         /* At this point, the preempt disable flag is still set, so we still
     have
123. 205            protection against all preemption.  */
124. 206
125. 207         /* Change thread priority to the new mutex priority-inheritance
     priority.  */
126. 208         thread_ptr -> tx_thread_priority =  new_priority; // 设置新的线程优先级
127. 209
128. 210         /* Determine how to setup the thread's preemption-threshold.  */
129. 211         if (thread_ptr -> tx_thread_user_preempt_threshold < new_priority) // 抢
     占阈值更新
130. 212         {
131. 213
132. 214             /* Change thread preemption-threshold to the user's preemption-
     threshold.  */
133. 215             thread_ptr -> tx_thread_preempt_threshold =  thread_ptr ->
     tx_thread_user_preempt_threshold;
134. 216         }
135. 217         else
136. 218         {
137. 219
138. 220             /* Change the thread preemption-threshold to the new threshold.  */
139. 221             thread_ptr -> tx_thread_preempt_threshold =  new_priority;
140. 222         }
```

```
223
224         /* Restore interrupts.  */
225         TX_RESTORE
226
227         /* Resume the thread with the new priority.  */
228         _tx_thread_system_resume(thread_ptr); // 唤醒线程，将线程加入就绪线程链表，会对禁止抢占计数器减1，前面加的2被减没了，但是_tx_mutex_priority_change被调用前，禁止抢占计数器都会先加1，所以_tx_thread_system_resume还是不能调度其他线程，不会切换线程，后续代码要重新对下一个要执行的线程进行计算
229 #endif
230
231         /* Optional processing extension.  */
232         TX_MUTEX_PRIORITY_CHANGE_EXTENSION
233
234 #ifndef TX_NOT_INTERRUPTABLE
235
236         /* Disable interrupts.  */
237         TX_DISABLE
238 #endif
239
240         /* Pickup the next thread to execute.  */
241         next_execute_ptr =  _tx_thread_execute_ptr; // 挂起/唤醒线程，允许中断过程可能有更高优先级线程就绪，获取当前下一个要执行的线程
242
243         /* Determine if this thread is not the next thread to execute.  */
244         if (thread_ptr != next_execute_ptr) // 被改变优先级的线程不是下一个需要执行的线程
245         {
246
247             /* Make sure the thread is still ready.  */
248             if (thread_ptr -> tx_thread_state == TX_READY) // _tx_thread_system_resume正常情况就会把线程设置为就绪状态，但是前面有解释延迟挂起，也就是在挂起线程中调用_tx_thread_system_resume之前，有其他操作挂起该线程并设置了延迟挂起，那么_tx_thread_system_resume就会挂起线程，而不会唤醒线程，因此这里还要再次判断线程状态
249             {
```

```
168. 250

169. 251                /* Now check and see if this thread has an equal or higher
     priority.  */

170. 252                if (thread_ptr -> tx_thread_priority <= next_execute_ptr ->
     tx_thread_priority) // 改变优先级的线程的优先级等于或者高于下一个需要执行的线程

171. 253                {

172. 254

173. 255                    /* Now determine if this thread was the previously executing
     thread.  */

174. 256                    if (thread_ptr == execute_ptr) // 如果thread_ptr在改变优先级
     之前就是下一个需要执行的线程，并且thread_ptr现在的优先级也不比next_execute_ptr低，那
     么应该继续选择thread_ptr作为下一个要执行的线程(释放互斥锁时，恢复低优先级，可能降低
     到次优先级就绪线程链表或者更低优先级线程启用了抢占，导致降低优先级的线程不能抢占线程
     next_execute_ptr)

175. 257                    {

176. 258

177. 259                        /* Yes, this thread was previously executing before we
     temporarily suspended and resumed

178. 260                           it in order to change the priority. A lower or same
     priority thread cannot be the next thread

179. 261                           to execute in this case since this thread really
     didn't suspend.  Simply reset the execute

180. 262                           pointer to this thread.  */

181. 263                        _tx_thread_execute_ptr =  thread_ptr; // 让thread_ptr线
     程继续执行

182. 264

183. 265                        /* Determine if we moved to a lower priority. If so,
     move the thread to the front of its priority list.  */

184. 266                        if (original_priority < new_priority) // 如果线程的优先
     级被降低了，thread_ptr可能添加到了就绪线程链表末尾，需要将thread_ptr移动到就绪线程链
     表表头(释放互斥锁优先级升高情况，暂时有些疑问，没看到怎么保证thread_ptr在链表表头，
     提交了一个issues到社区，等待社区回复!!!)

185. 267                        {

186. 268

187. 269                            /* Ensure that this thread is placed at the front of
     the priority list.  */

188. 270                            _tx_thread_priority_list[thread_ptr ->
     tx_thread_priority] =  thread_ptr; // 就绪线程链表表头指向thread_ptr(等价将
     thread_ptr移动到表头)

189. 271                        }
```

```
190. 272                    }

191. 273                }

192. 274                else // 1、提高优先级过程有更高优先级线程被中断服务程序唤醒，next_execute_ptr为被唤醒的高优先级线程 2、线程优先级降低，降低后已经不是最高优先级

193. 275                {

194. 276

195. 277                    /* Now determine if this thread's preemption-threshold needs to be enforced.  */

196. 278                    if (thread_ptr -> tx_thread_preempt_threshold < thread_ptr -> tx_thread_priority) // thread_ptr启用了抢占，需要检查thread_ptr是否可以抢占next_execute_ptr(如果thread_ptr是当前释放互斥锁的线程，thread_ptr本来就是抢占了一些其他高优先级的线程，因为挂起恢复操作，内核选择了其他被抢占的高优先级线程执行，那么是不合理的，需要让thread_ptr继续抢占其他高优先级线程；如果有更高优先级线程被唤醒，且优先级高于thread_ptr的抢占阈值，那么thread_ptr需要标记被抢占)

197. 279                    {

198. 280

199. 281                        /* Yes, preemption-threshold is in force for this thread. */

200. 282

201. 283                        /* Compare the next thread to execute thread's priority against the thread's preemption-threshold.  */

202. 284                        if (thread_ptr -> tx_thread_preempt_threshold <= next_execute_ptr -> tx_thread_priority) // thread_ptr的抢占阈值高于next_execute_ptr，thread_ptr抢占next_execute_ptr

203. 285                        {

204. 286

205. 287                            /* We must swap execute pointers to enforce the preemption-threshold of a thread coming out of

206. 288                               priority inheritance.  */

207. 289                            _tx_thread_execute_ptr =  thread_ptr;

208. 290

209. 291                            /* Determine if we moved to a lower priority. If so, move the thread to the front of its priority list.  */

210. 292                            if (original_priority < new_priority) // 这里升高优先级也需要考虑，还有疑问，等待社区回复!!!

211. 293                            {

212. 294

213. 295                                /* Ensure that this thread is placed at the front of the priority list.  */
```

```
296                                 _tx_thread_priority_list[thread_ptr ->
tx_thread_priority] =  thread_ptr;

297                             }

298                         }

299

300 #ifndef TX_DISABLE_PREEMPTION_THRESHOLD

301

302                         else // 下面的抢占标记似乎应该保证thread_ptr在链表表头，
这个问题待社区回复(阅读threadx-6.1.2代码的时候，发现这个标记也有问题，然后查看官网最
新代码已经修复了，写文章时顺道改了，旧的代码标记的是next_execute_ptr，实际是
thread_ptr被抢占了，不是next_execute_ptr被抢占)

303                         {

304

305                             /* In this case, we need to mark the preempted map
to indicate a thread executed above the

306                                preemption-threshold.  */

307

308 #if TX_MAX_PRIORITIES > 32

309

310                             /* Calculate the index into the bit map array.  */

311                             map_index =  (thread_ptr -> tx_thread_priority)/
((UINT) 32);

312

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

314                             TX_DIV32_BIT_SET(thread_ptr -> tx_thread_priority,
priority_bit)

315                             _tx_thread_preempted_map_active =
_tx_thread_preempted_map_active | priority_bit;

316 #endif

317

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

319                             TX_MOD32_BIT_SET(thread_ptr -> tx_thread_priority,
priority_bit)

320                             _tx_thread_preempted_maps[MAP_INDEX] =
_tx_thread_preempted_maps[MAP_INDEX] | priority_bit;

321                         }
```

```
240. 322 #endif
241. 323                     }
242. 324                 }
243. 325             }
244. 326         }
245. 327
246. 328 #ifndef TX_NOT_INTERRUPTABLE
247. 329
248. 330         /* Restore interrupts.  */
249. 331         TX_RESTORE
250. 332 #endif
251. 333     }
252. 334 }
253. 335
```

3、互斥锁释放_tx_mutex_put

互斥锁释放与互斥锁申请类似，因为互斥锁是可多次获取的，因此释放互斥锁也是要对互斥锁计数器减1，如果为0，才是真正释放互斥锁，因为存在继承优先级，释放互斥锁线程当前运行的优先级并不一定是创建时的优先级，如果没有占用其他互斥锁(不需要继承互斥锁优先级)，那么就需要恢复创建时的优先级，如果还有占用其他线程，那么需要继承其他等待互斥锁线程的最高优先级，调整优先级跟获取互斥锁调整优先级一样的，前面小结已经介绍；

继承优先级，释放互斥锁时，是将互斥锁直接给等待互斥锁的最高优先级线程，也就是优先级高的线程先获取到互斥锁。

_tx_mutex_put实现代码如下：

```
 1. 047 UINT  _tx_mutex_put(TX_MUTEX *mutex_ptr)
 2. 048 {
 3. 049
 4. 050 TX_INTERRUPT_SAVE_AREA
 5. 051
 6. 052 TX_THREAD       *thread_ptr;
 7. 053 TX_THREAD       *old_owner;
 8. 054 UINT            old_priority;
 9. 055 UINT            status;
10. 056 TX_MUTEX        *next_mutex;
11. 057 TX_MUTEX        *previous_mutex;
12. 058 UINT            owned_count;
13. 059 UINT            suspended_count;
14. 060 TX_THREAD       *current_thread;
15. 061 TX_THREAD       *next_thread;
16. 062 TX_THREAD       *previous_thread;
17. 063 TX_THREAD       *suspended_thread;
18. 064 UINT            inheritance_priority;
19. 065
20. 066
21. 067     /* Setup status to indicate the processing is not complete.  */
22. 068     status =  TX_NOT_DONE;
23. 069
24. 070     /* Disable interrupts to put an instance back to the mutex.  */
25. 071     TX_DISABLE
26. 072
27. 073 #ifdef TX_MUTEX_ENABLE_PERFORMANCE_INFO
28. 074
29. 075     /* Increment the total mutex put counter.  */
30. 076     _tx_mutex_performance_put_count++;
31. 077
```

```
078     /* Increment the number of attempts to put this mutex.  */
079     mutex_ptr -> tx_mutex_performance_put_count++;
080 #endif
081
082     /* If trace is enabled, insert this event into the trace buffer.  */
083     TX_TRACE_IN_LINE_INSERT(TX_TRACE_MUTEX_PUT, mutex_ptr,
    TX_POINTER_TO_ULONG_CONVERT(mutex_ptr -> tx_mutex_owner), mutex_ptr ->
    tx_mutex_ownership_count, TX_POINTER_TO_ULONG_CONVERT(&old_priority),
    TX_TRACE_MUTEX_EVENTS)
084
085     /* Log this kernel call.  */
086     TX_EL_MUTEX_PUT_INSERT
087
088     /* Determine if this mutex is owned.  */
089     if (mutex_ptr -> tx_mutex_ownership_count != ((UINT) 0)) // 互斥锁计数器不为
    0才被占用
090     {
091
092         /* Pickup the owning thread pointer.  */
093         thread_ptr =  mutex_ptr -> tx_mutex_owner; // 获取占用互斥锁的线程
094
095         /* Pickup thread pointer.  */
096         TX_THREAD_GET_CURRENT(current_thread) // 获取当前线程
097
098         /* Check to see if the mutex is owned by the calling thread.  */
099         if (mutex_ptr -> tx_mutex_owner != current_thread) // 如果占用互斥锁的线
    程不是当前线程，设置状态为TX_NOT_OWNED，不能释放别的线程占用的互斥锁，返回
100         {
101
102             /* Determine if the preempt disable flag is set, indicating that
103                the caller is not the application but from ThreadX. In such
104                cases, the thread mutex owner does not need to match.  */
105             if (_tx_thread_preempt_disable == ((UINT) 0)) //
    _tx_thread_preempt_disable为0表示是应用程序调用，应用程序不能释放其他线程占用的互斥
    锁
```

```
106             {
107
108                 /* Invalid mutex release.  */
109
110                 /* Restore interrupts.  */
111                 TX_RESTORE
112
113                 /* Caller does not own the mutex.  */
114                 status =  TX_NOT_OWNED;
115             }
116         }
117
118         /* Determine if we should continue.  */
119         if (status == TX_NOT_DONE)
120         {
121
122             /* Decrement the mutex ownership count.  */
123             mutex_ptr -> tx_mutex_ownership_count--; // 互斥锁计数器减1
124
125             /* Determine if the mutex is still owned by the current thread.  */
126             if (mutex_ptr -> tx_mutex_ownership_count != ((UINT) 0)) // 互斥锁嵌套占用，没有真正释放，设置状态为TX_SUCCESS，返回即可
127             {
128
129                 /* Restore interrupts.  */
130                 TX_RESTORE
131
132                 /* Mutex is still owned, just return successful status.  */
133                 status =  TX_SUCCESS;
134             }
135             else // 互斥锁被释放
136             {
```

```
137
138                 /* Check for a NULL thread pointer, which can only happen during initialization.   */
139                 if (thread_ptr == TX_NULL) // 内核初始化过程释放互斥锁，不需要处理
140                 {
141
142                     /* Restore interrupts.  */
143                     TX_RESTORE
144
145                     /* Mutex is now available, return successful status.  */
146                     status =  TX_SUCCESS;
147                 }
148                 else // 线程释放互斥锁
149                 {
150
151                     /* The mutex is now available.   */
152
153                     /* Remove this mutex from the owned mutex list.  */
154
155                     /* Decrement the ownership count.  */
156                     thread_ptr -> tx_thread_owned_mutex_count--; // 线程占用的互斥锁个数减1
157
158                     /* Determine if this mutex was the only one on the list.  */
159                     if (thread_ptr -> tx_thread_owned_mutex_count == ((UINT) 0)) // 占用的互斥锁个数为0，占用互斥锁链表tx_thread_owned_mutex_list设置为0即可
160                     {
161
162                         /* Yes, the list is empty.  Simply set the head pointer to NULL.  */
163                         thread_ptr -> tx_thread_owned_mutex_list =  TX_NULL;
164                     }
165                     else // 将释放的互斥锁从tx_thread_owned_mutex_list删除
```

```
120. 166                    {
121. 167
122. 168                        /* No, there are more mutexes on the list.  */
123. 169
124. 170                        /* Link-up the neighbors.  */
125. 171                        next_mutex =                            mutex_ptr ->
     tx_mutex_owned_next;
126. 172                        previous_mutex =                        mutex_ptr ->
     tx_mutex_owned_previous;
127. 173                        next_mutex -> tx_mutex_owned_previous =  previous_mutex;
128. 174                        previous_mutex -> tx_mutex_owned_next =  next_mutex;
129. 175
130. 176                        /* See if we have to update the created list head
     pointer.  */
131. 177                        if (thread_ptr -> tx_thread_owned_mutex_list ==
     mutex_ptr)
132. 178                        {
133. 179
134. 180                            /* Yes, move the head pointer to the next link. */
135. 181                            thread_ptr -> tx_thread_owned_mutex_list =
     next_mutex;
136. 182                        }
137. 183                    }
138. 184
139. 185                    /* Determine if the simple, non-suspension, non-priority
     inheritance case is present.  */
140. 186                    if (mutex_ptr -> tx_mutex_suspension_list == TX_NULL) // 没
     有等待互斥锁的线程
141. 187                    {
142. 188
143. 189                        /* Is this a priority inheritance mutex?  */
144. 190                        if (mutex_ptr -> tx_mutex_inherit == TX_FALSE) // 如果没
     有继承优先级(占用互斥锁的线程不会动态调整优先级)，返回即可
145. 191                        {
146. 192
```

```
147. 193                            /* Yes, we are done - set the mutex owner to NULL.
     */
148. 194                            mutex_ptr -> tx_mutex_owner =  TX_NULL;
149. 195
150. 196                            /* Restore interrupts.  */
151. 197                            TX_RESTORE
152. 198
153. 199                            /* Mutex is now available, return successful status.
     */
154. 200                            status =  TX_SUCCESS;
155. 201                        }
156. 202                    }
157. 203
158. 204                    /* Determine if the processing is complete.  */
159. 205                    if (status == TX_NOT_DONE) // 有等待互斥锁的线程或者有继承优
     先级(需要还原优先级)
160. 206                    {
161. 207
162. 208                        /* Initialize original owner and thread priority.  */
163. 209                        old_owner =      TX_NULL;
164. 210                        old_priority =   thread_ptr -> tx_thread_user_priority;
165. 211
166. 212                        /* Does this mutex support priority inheritance?  */
167. 213                        if (mutex_ptr -> tx_mutex_inherit == TX_TRUE) // 互斥锁
     支持继承优先级(if分支获取继承的优先级，如果继承的优先级比线程创建时的优先级还低，那
     么恢复到线程创建时的优先级(先记录，if后面恢复))
168. 214                        {
169. 215
170. 216 #ifndef TX_NOT_INTERRUPTABLE
171. 217
172. 218                            /* Temporarily disable preemption.  */
173. 219                            _tx_thread_preempt_disable++; // 禁止抢占
174. 220
175. 221                            /* Restore interrupts.  */
```

```
176. 222                            TX_RESTORE // 允许中断(抢占被禁止了，中断服务程序不会操作互斥锁相关的数据，所以可以允许中断)

177. 223 #endif

178. 224

179. 225                            /* Default the inheritance priority to disabled.  */

180. 226                            inheritance_priority =  ((UINT) TX_MAX_PRIORITIES); // 继承的优先级

181. 227

182. 228                            /* Search the owned mutexes for this thread to determine the highest priority for this

183. 229                               former mutex owner to return to.  */

184. 230                            next_mutex =  thread_ptr -> tx_thread_owned_mutex_list; // 下一个互斥锁

185. 231                            while (next_mutex != TX_NULL) // 遍历互斥锁链表(查找当前线程需要继承的最高优先级)

186. 232                            {

187. 233

188. 234                                /* Does this mutex support priority inheritance?  */

189. 235                                if (next_mutex -> tx_mutex_inherit == TX_TRUE) // next_mutex支持继承优先级

190. 236                                {

191. 237

192. 238                                    /* Determine if highest priority field of the mutex is higher than the priority to

193. 239                                       restore.  */

194. 240                                    if (next_mutex -> tx_mutex_highest_priority_waiting < inheritance_priority) // next_mutex等待互斥锁的线程的最高优先级高于inheritance_priority

195. 241                                    {

196. 242

197. 243                                        /* Use this priority to return releasing thread to.  */

198. 244                                        inheritance_priority =   next_mutex -> tx_mutex_highest_priority_waiting; // inheritance_priority更新为继承的最高优先级

199. 245                                    }

200. 246                                }

201. 247
```

```
248                                 /* Move mutex pointer to the next mutex in the
    list.  */
249                                 next_mutex =  next_mutex -> tx_mutex_owned_next;
250
251                                 /* Are we at the end of the list?  */
252                                 if (next_mutex == thread_ptr ->
    tx_thread_owned_mutex_list)
253                                 {
254
255                                     /* Yes, set the next mutex to NULL.  */
256                                     next_mutex =  TX_NULL;
257                                 }
258                             }
259
260 #ifndef TX_NOT_INTERRUPTABLE
261
262                             /* Disable interrupts.  */
263                             TX_DISABLE // 关闭中断(可能中断会影响后面的代码或者
    性能，并且后面很快就会允许中断，所以禁止中断)
264
265                             /* Undo the temporarily preemption disable.  */
266                             _tx_thread_preempt_disable--; // 取消临时的禁止抢占
267 #endif
268
269                             /* Set the inherit priority to that of the highest
    priority thread waiting on the mutex.  */
270                             thread_ptr -> tx_thread_inherit_priority =
    inheritance_priority; // 设置线程继承优先级
271
272                             /* Determine if the inheritance priority is less
    than the default old priority.  */
273                             if (inheritance_priority < old_priority) // 如果继承
    的优先级高于线程创建时指定的优先级，那么使用继承优先级做为线程的优先级(old_priority
    在后面会用于调整线程的优先级)，否则释放互斥锁锁后恢复到线程创建时的优先级
274                             {
275
```

```
230. 276                                /* Yes, update the old priority.  */
231. 277                                old_priority =  inheritance_priority;
232. 278                            }
233. 279                        }
234. 280
235. 281                        /* Determine if priority inheritance is in effect and
     there are one or more
236. 282                           threads suspended on the mutex.  */
237. 283                        if (mutex_ptr -> tx_mutex_suspended_count > ((UINT) 1))
     // 有多于1个线程等待互斥锁mutex_ptr，并且互斥锁有优先级继承，那么需要唤醒最高优先级的等待线程
238. 284                        {
239. 285
240. 286                            /* Is priority inheritance in effect?  */
241. 287                            if (mutex_ptr -> tx_mutex_inherit == TX_TRUE)
242. 288                            {
243. 289
244. 290                                /* Yes, this code is simply to ensure the
     highest priority thread is positioned
245. 291                                   at the front of the suspension list.  */
246. 292
247. 293 #ifndef TX_NOT_INTERRUPTABLE
248. 294
249. 295                                /* Temporarily disable preemption.  */
250. 296                                _tx_thread_preempt_disable++; // 禁止抢占(后面对等待互斥锁的线程链表进行处理，如果允许抢占的话，有等待互斥锁的线程超时的话，超时回调函数也会操作链表，就保证不了对链表的互斥操作)
251. 297
252. 298                                /* Restore interrupts.  */
253. 299                                TX_RESTORE // 允许中断(后面把等待互斥锁的最高优先级线程移动到链表前面需要一些时间，需要允许中断，否则定时器中断就不能及时响应)
254. 300 #endif
255. 301
256. 302                                /* Call the mutex prioritize processing to
     ensure the
257. 303                                   highest priority thread is resumed.  */
```

```
304 #ifdef TX_MISRA_ENABLE
305                                 do
306                                 {
307                                     status =  _tx_mutex_prioritize(mutex_ptr);
308                                 } while (status != TX_SUCCESS);
309 #else
310                                 _tx_mutex_prioritize(mutex_ptr); // 将等待互斥锁线程的最高优先级线程移动到等待链表前面(优先级高的线程先获取到互斥锁)
311 #endif
312
313                                 /* At this point, the highest priority thread is at the
314                                    front of the suspension list.  */
315
316                                 /* Optional processing extension.  */
317                                 TX_MUTEX_PUT_EXTENSION_1
318
319 #ifndef TX_NOT_INTERRUPTABLE
320
321                                 /* Disable interrupts.  */
322                                 TX_DISABLE
323
324                                 /* Back off the preemption disable.  */
325                                 _tx_thread_preempt_disable--;
326 #endif
327                             }
328                         }
329
330                         /* Now determine if there are any threads still waiting on the mutex.  */
331                         if (mutex_ptr -> tx_mutex_suspension_list == TX_NULL) // 检查是否有线程仍然在等待互斥锁(前面检查过tx_mutex_suspension_list不为空，目前互斥锁等待超时是在超时线程里面处理，如果互斥锁超时在中断服务程序里面处理，那么tx_mutex_suspension_list倒可能被修改，暂时没看到其他场景，正常情况这里不会为空，先略过...)
```

```
286. 332                        {
287. 333
288. 334                            /* No, there are no longer any threads waiting on
     the mutex.  */
289. 335
290. 336 #ifndef TX_NOT_INTERRUPTABLE
291. 337
292. 338                            /* Temporarily disable preemption.  */
293. 339                            _tx_thread_preempt_disable++;
294. 340
295. 341                            /* Restore interrupts.  */
296. 342                            TX_RESTORE
297. 343 #endif
298. 344
299. 345                            /* Mutex is not owned, but it is possible that a
     thread that
300. 346                               caused a priority inheritance to occur is no
     longer waiting
301. 347                               on the mutex.  */
302. 348
303. 349                            /* Setup the highest priority waiting thread.  */
304. 350                            mutex_ptr -> tx_mutex_highest_priority_waiting =
     (UINT) TX_MAX_PRIORITIES; // 没有等待互斥锁的线程，等待互斥锁线程的最高优先级设置为
     默认优先级TX_MAX_PRIORITIES
305. 351
306. 352                            /* Determine if we need to restore priority.  */
307. 353                            if ((mutex_ptr -> tx_mutex_owner) ->
     tx_thread_priority != old_priority) // (mutex_ptr -> tx_mutex_owner) ->
     tx_thread_priority记录占用互斥锁线程运行时的优先级，old_priority为前面检查出来的等待
     互斥锁线程的最高优先级或者线程创建时的优先级，也就是释放互斥锁后，线程该恢复的优先
     级，如果当前的优先级与要恢复的优先级不同，那么就调用_tx_mutex_priority_change调整/恢
     复线程优先级，mutex_ptr -> tx_mutex_owner前面已经检查了，就是current_thread
308. 354                            {
309. 355
310. 356                                /* Yes, restore the priority of thread.  */
311. 357                                _tx_mutex_priority_change(mutex_ptr ->
     tx_mutex_owner, old_priority); // 恢复线程创建时的优先级或者继承的最高优先级
```

```
358                             }
359
360 #ifndef TX_NOT_INTERRUPTABLE
361
362                             /* Disable interrupts again.  */
363                             TX_DISABLE
364
365                             /* Back off the preemption disable.  */
366                             _tx_thread_preempt_disable--;
367 #endif
368
369                             /* Set the mutex owner to NULL.  */
370                             mutex_ptr -> tx_mutex_owner =  TX_NULL;
371
372                             /* Restore interrupts.  */
373                             TX_RESTORE
374
375                             /* Check for preemption.  */
376                             _tx_thread_system_preempt_check(); // 这里基本已经释放完了互斥锁，后续也没有其他操作要进行，因为前面禁止了抢占，释放互斥锁的上一级函数不会检查抢占，禁止抢占的过程中，可能有更高优先级线程就绪，所以在这里检查抢占，如果有被抢占的话，那么要重新调度
377
378                             /* Set status to success.  */
379                             status =  TX_SUCCESS;
380                         }
381                         else // 有线程等待互斥锁
382                         {
383
384                             /* Pickup the thread at the front of the suspension list.  */
385                             thread_ptr =  mutex_ptr -> tx_mutex_suspension_list; // 获取等待互斥锁的最高优先级线程(前面已经将最高优先级线程移动到表头了)
386
```

```
387                             /* Save the previous ownership information, if
    inheritance is
388                                in effect.  */
389                             if (mutex_ptr -> tx_mutex_inherit == TX_TRUE) // 继
    承优先级
390                             {
391
392                                 /* Remember the old mutex owner.  */
393                                 old_owner =  mutex_ptr -> tx_mutex_owner; // 记
    录旧的占用互斥锁的线程
394
395                                 /* Setup owner thread priority information.  */
396                                 mutex_ptr -> tx_mutex_original_priority =
    thread_ptr -> tx_thread_priority; // 记录thread_ptr的原始优先级(thread_ptr即将获得互
    斥锁，因为继承优先级的关系，thread_ptr可能会改变优先级)
397
398                                 /* Setup the highest priority waiting thread.
    */
399                                 mutex_ptr -> tx_mutex_highest_priority_waiting =
    (UINT) TX_MAX_PRIORITIES;
400                             }
401
402                             /* Determine how many mutexes are owned by this
    thread.  */
403                             owned_count =  thread_ptr ->
    tx_thread_owned_mutex_count; // thread_ptr之前占用多少互斥锁(互斥锁嵌套，线程可能占
    用多个互斥锁)
404
405                             /* Determine if this thread owns any other mutexes
    that have priority inheritance.  */
406                             if (owned_count == ((UINT) 0)) // thread_ptr没有占用
    其他线程，那么当前互斥锁就是该线程唯一获得的互斥锁，一个互斥锁组成一个链表(线程占用
    互斥锁的链表tx_thread_owned_mutex_list)
407                             {
408
409                                 /* The owned mutex list is empty.  Add mutex to
    empty list.  */
410                                 thread_ptr -> tx_thread_owned_mutex_list =
    mutex_ptr;
```

```
411                                 mutex_ptr -> tx_mutex_owned_next =
    mutex_ptr;

412                                 mutex_ptr -> tx_mutex_owned_previous =
    mutex_ptr;

413                             }

414                             else // 将互斥锁加入旧的占用互斥锁链表
    tx_thread_owned_mutex_list

415                             {

416

417                                 /* Non-empty list. Link up the mutex.  */

418

419                                 /* Pickup tail pointer.  */

420                                 next_mutex =
    thread_ptr -> tx_thread_owned_mutex_list;

421                                 previous_mutex =
    next_mutex -> tx_mutex_owned_previous; // previous_mutex第一个节点的前一个节点，也就
    是尾节点

422

423                                 /* Place the owned mutex in the list.  */

424                                 next_mutex -> tx_mutex_owned_previous =
    mutex_ptr;

425                                 previous_mutex -> tx_mutex_owned_next =
    mutex_ptr;

426

427                                 /* Setup this mutex's next and previous created
    links.  */

428                                 mutex_ptr -> tx_mutex_owned_previous =
    previous_mutex;

429                                 mutex_ptr -> tx_mutex_owned_next =
    next_mutex;

430                             }

431

432                             /* Increment the number of mutexes owned counter.
    */

433                             thread_ptr -> tx_thread_owned_mutex_count =
    owned_count + ((UINT) 1); // thread_ptr占用互斥锁个数加1(释放互斥锁的时候，直接将互
    斥锁给thread_ptr，不需要唤醒所有等待互斥锁的线程，让这些线程去抢占互斥锁)

434
```

```
389. 435                            /* Mark the Mutex as owned and fill in the
     corresponding information.  */

390. 436                            mutex_ptr -> tx_mutex_ownership_count =  (UINT) 1;
     // mutex_ptr第一次获取到互斥锁mutex_ptr

391. 437                            mutex_ptr -> tx_mutex_owner =            thread_ptr;
     // 互斥锁mutex_ptr被线程thread_ptr占用

392. 438

393. 439                            /* Remove the suspended thread from the list.  */

394. 440

395. 441                            /* Decrement the suspension count.  */

396. 442                            mutex_ptr -> tx_mutex_suspended_count--; //
     thread_ptr已经获取到了互斥锁，等待互斥锁计数器减1

397. 443

398. 444                            /* Pickup the suspended count.  */

399. 445                            suspended_count =  mutex_ptr ->
     tx_mutex_suspended_count;

400. 446

401. 447                            /* See if this is the only suspended thread on the
     list.  */

402. 448                            if (suspended_count == TX_NO_SUSPENSIONS) // 没有线
     程等待互斥锁了(原来就只有thread_ptr等待，现在thread_ptr已经获取到互斥锁，等待链表设
     置为空即可)

403. 449                            {

404. 450

405. 451                                /* Yes, the only suspended thread.  */

406. 452

407. 453                                /* Update the head pointer.  */

408. 454                                mutex_ptr -> tx_mutex_suspension_list =
     TX_NULL;

409. 455                            }

410. 456                            else // 等待互斥锁的链表还有其他线程，将thread_ptr从
     等待链表删除(前面是将互斥锁释放给等待链表表头线程，那么这里删除表头结点即可，表头结
     点也就是thread_ptr)

411. 457                            {

412. 458

413. 459                                /* At least one more thread is on the same
     expiration list.  */

414. 460
```

```
415. 461                                    /* Update the list head pointer.  */
416. 462                                    next_thread =
     thread_ptr -> tx_thread_suspended_next;
417. 463                                    mutex_ptr -> tx_mutex_suspension_list =
     next_thread;
418. 464
419. 465                                    /* Update the links of the adjacent threads.  */
420. 466                                    previous_thread =
     thread_ptr -> tx_thread_suspended_previous;
421. 467                                    next_thread -> tx_thread_suspended_previous =
     previous_thread;
422. 468                                    previous_thread -> tx_thread_suspended_next =
     next_thread;
423. 469                                }
424. 470
425. 471                                /* Prepare for resumption of the first thread.  */
426. 472
427. 473                                /* Clear cleanup routine to avoid timeout.  */
428. 474                                thread_ptr -> tx_thread_suspend_cleanup =  TX_NULL;
     // 清空thread_ptr的tx_thread_suspend_cleanup(tx_thread_suspend_cleanup主要用于等待互
     斥锁超时，唤醒等待线程)
429. 475
430. 476                                /* Put return status into the thread control block.
     */
431. 477                                thread_ptr -> tx_thread_suspend_status =
     TX_SUCCESS;
432. 478
433. 479 #ifdef TX_NOT_INTERRUPTABLE
434. 480
435. 481                                /* Determine if priority inheritance is enabled for
     this mutex.  */
436. 482                                if (mutex_ptr -> tx_mutex_inherit == TX_TRUE)
437. 483                                {
438. 484
439. 485                                    /* Yes, priority inheritance is requested.  */
440. 486
```

```
441. 487                                    /* Determine if there are any more threads still
     suspended on the mutex.  */
442. 488                                    if (mutex_ptr -> tx_mutex_suspended_count !=
     ((ULONG) 0))
443. 489                                    {
444. 490
445. 491                                        /* Determine if there are more than one
     thread suspended on the mutex.  */
446. 492                                        if (mutex_ptr -> tx_mutex_suspended_count >
     ((ULONG) 1))
447. 493                                        {
448. 494
449. 495                                            /* If so, prioritize the list so the
     highest priority thread is placed at the
450. 496                                               front of the suspension list.  */
451. 497 #ifdef TX_MISRA_ENABLE
452. 498                                            do
453. 499                                            {
454. 500                                                status =
     _tx_mutex_prioritize(mutex_ptr);
455. 501                                            } while (status != TX_SUCCESS);
456. 502 #else
457. 503                                            _tx_mutex_prioritize(mutex_ptr);
458. 504 #endif
459. 505                                        }
460. 506
461. 507                                        /* Now, pickup the list head and set the
     priority.  */
462. 508
463. 509                                        /* Determine if there still are threads
     suspended for this mutex.  */
464. 510                                        suspended_thread =  mutex_ptr ->
     tx_mutex_suspension_list;
465. 511                                        if (suspended_thread != TX_NULL)
466. 512                                        {
467. 513
```

```
468. 514                                        /* Setup the highest priority thread
     waiting on this mutex.  */

469. 515                                        mutex_ptr ->
     tx_mutex_highest_priority_waiting =  suspended_thread -> tx_thread_priority;

470. 516                                    }

471. 517                                }

472. 518

473. 519                                /* Restore previous priority needs to be
     restored after priority

474. 520                                   inheritance.  */

475. 521

476. 522                                /* Determine if we need to restore priority.  */

477. 523                                if (old_owner -> tx_thread_priority !=
     old_priority)

478. 524                                {

479. 525

480. 526                                    /* Restore priority of thread.  */

481. 527                                    _tx_mutex_priority_change(old_owner,
     old_priority);

482. 528                                }

483. 529                            }

484. 530

485. 531                            /* Resume the thread!  */

486. 532                            _tx_thread_system_ni_resume(thread_ptr);

487. 533

488. 534                            /* Restore interrupts.  */

489. 535                            TX_RESTORE

490. 536 #else

491. 537

492. 538                            /* Temporarily disable preemption.  */

493. 539                            _tx_thread_preempt_disable++; // 禁止抢占
     (_tx_mutex_prioritize及其他操作耗时比较多，不能关中断)

494. 540

495. 541                            /* Restore interrupts.  */

496. 542                            TX_RESTORE
```

```
497. 543

498. 544                            /* Determine if priority inheritance is enabled for
     this mutex.  */

499. 545                            if (mutex_ptr -> tx_mutex_inherit == TX_TRUE) // if
     分支除了更新mutex_ptr等待线程最高优先级外，还恢复了释放互斥锁线程的优先级，一个判断
     处理两件事情

500. 546                            {

501. 547

502. 548                                /* Yes, priority inheritance is requested.  */

503. 549

504. 550                                /* Determine if there are any more threads still
     suspended on the mutex.  */

505. 551                                if (mutex_ptr -> tx_mutex_suspended_count !=
     TX_NO_SUSPENSIONS) // if分支获取mutex_ptr等待线程链表最高优先级，更新
     tx_mutex_highest_priority_waiting

506. 552                                {

507. 553

508. 554                                    /* Prioritize the list so the highest
     priority thread is placed at the

509. 555                                       front of the suspension list.  */

510. 556 #ifdef TX_MISRA_ENABLE

511. 557                                    do

512. 558                                    {

513. 559                                        status =
     _tx_mutex_prioritize(mutex_ptr);

514. 560                                    } while (status != TX_SUCCESS);

515. 561 #else

516. 562                                    _tx_mutex_prioritize(mutex_ptr);

517. 563 #endif

518. 564

519. 565                                    /* Now, pickup the list head and set the
     priority.  */

520. 566

521. 567                                    /* Optional processing extension.  */

522. 568                                    TX_MUTEX_PUT_EXTENSION_2

523. 569
```

```
524. 570                                    /* Disable interrupts.  */
525. 571                                    TX_DISABLE
526. 572
527. 573                                    /* Determine if there still are threads
     suspended for this mutex.  */
528. 574                                    suspended_thread =  mutex_ptr ->
     tx_mutex_suspension_list;
529. 575                                    if (suspended_thread != TX_NULL)
530. 576                                    {
531. 577
532. 578                                        /* Setup the highest priority thread
     waiting on this mutex.  */
533. 579                                        mutex_ptr ->
     tx_mutex_highest_priority_waiting =  suspended_thread -> tx_thread_priority; // 更新
     tx_mutex_highest_priority_waiting
534. 580                                    }
535. 581
536. 582                                    /* Restore interrupts.  */
537. 583                                    TX_RESTORE
538. 584                                }
539. 585
540. 586                                /* Restore previous priority needs to be
     restored after priority
541. 587                                   inheritance.  */
542. 588
543. 589                                /* Is the priority different?  */
544. 590                                if (old_owner -> tx_thread_priority !=
     old_priority) // old_priority前面判断需要恢复的优先级(释放互斥锁后恢复到线程创建时优
     先级或者互斥锁继承的优先级，如果两个优先级一样，那么不需要更新)
545. 591                                {
546. 592
547. 593                                    /* Restore the priority of thread.  */
548. 594                                    _tx_mutex_priority_change(old_owner,
     old_priority); // 恢复释放互斥锁线程的优先级
549. 595                                }
550. 596                            }
```

```
597
598                             /* Resume thread.  */
599                             _tx_thread_system_resume(thread_ptr); // 唤醒thread_ptr线程(删除thread_ptr的等待互斥锁超时定时器，唤醒thread_ptr线程，检查抢占等，如果有抢占，可能发生线程切换...)
600 #endif
601
602                             /* Return a successful status.  */
603                             status =  TX_SUCCESS;
604                         }
605                     }
606                 }
607             }
608         }
609     }
610     else // 当前线程没有占用互斥锁，不能释放，返回TX_NOT_OWNED
611     {
612
613         /* Restore interrupts.  */
614         TX_RESTORE
615
616         /* Caller does not own the mutex.  */
617         status =  TX_NOT_OWNED;
618     }
619
620     /* Return the completion status.  */
621     return(status);
622 }
```