

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

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

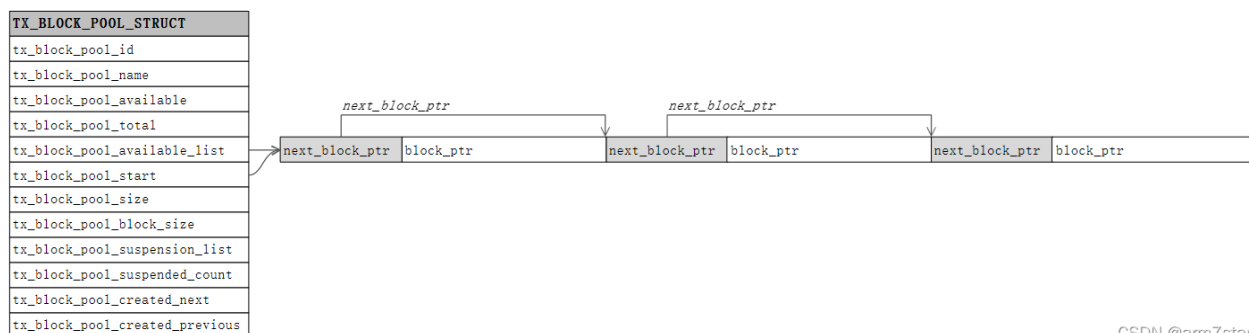
ThreadX内核与 $\mu\text{C}$  / OS-II、NucleusPlus内核一样，都支持固定大小的内存块管理，也支持不固定大小的内存块管理。

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

## 1、block内存管理

### 1.1、block内存块介绍

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



CSDN @arm7star

### 1.2、block内存申请\_tx\_block\_allocate

申请内存主要是获取可用内存块，主要过程是：

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

\_tx\_block\_allocate申请内存代码如下：

```

1. 080 UINT _tx_block_allocate(TX_BLOCK_POOL *pool_ptr, VOID **block_ptr, ULONG
    wait_option)

2. 081 {

3. 082

4. 083 TX_INTERRUPT_SAVE_AREA

5. 084

6. 085 UINT                status;

7. 086 TX_THREAD           *thread_ptr;

8. 087 UCHAR               *work_ptr;

9. 088 UCHAR               *temp_ptr;

10. 089 UCHAR              **next_block_ptr;

11. 090 UCHAR              **return_ptr;

12. 091 UINT               suspended_count;

13. 092 TX_THREAD          *next_thread;

14. 093 TX_THREAD          *previous_thread;

15. 094 #ifdef TX_ENABLE_EVENT_TRACE

16. 095 TX_TRACE_BUFFER_ENTRY *entry_ptr;

17. 096 ULONG              time_stamp = ((ULONG) 0);

18. 097 #endif

19. 098 #ifdef TX_ENABLE_EVENT_LOGGING

20. 099 UCHAR               *log_entry_ptr;

21. 100 ULONG              upper_tbu;

22. 101 ULONG              lower_tbu;

23. 102 #endif

24. 103

25. 104

26. 105     /* Disable interrupts to get a block from the pool. */

27. 106     TX_DISABLE

28. 107

29. 108 #ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO

30. 109

31. 110     /* Increment the total allocations counter. */

```

```

32. 111     _tx_block_pool_performance_allocate_count++;

33. 112

34. 113     /* Increment the number of allocations on this pool. */

35. 114     pool_ptr -> tx_block_pool_performance_allocate_count++;

36. 115 #endif

37. 116

38. 117 #ifdef TX_ENABLE_EVENT_TRACE

39. 118

40. 119     /* If trace is enabled, save the current event pointer. */

41. 120     entry_ptr = _tx_trace_buffer_current_ptr;

42. 121

43. 122     /* If trace is enabled, insert this event into the trace buffer. */

44. 123     TX_TRACE_IN_LINE_INSERT(TX_TRACE_BLOCK_ALLOCATE, pool_ptr, 0, wait_option,
        pool_ptr -> tx_block_pool_available, TX_TRACE_BLOCK_POOL_EVENTS)

45. 124

46. 125     /* Save the time stamp for later comparison to verify that

47. 126         the event hasn't been overwritten by the time the allocate

48. 127         call succeeds. */

49. 128     if (entry_ptr != TX_NULL)

50. 129     {

51. 130

52. 131         time_stamp = entry_ptr -> tx_trace_buffer_entry_time_stamp;

53. 132     }

54. 133 #endif

55. 134

56. 135 #ifdef TX_ENABLE_EVENT_LOGGING

57. 136     log_entry_ptr = *(UCHAR **) _tx_el_current_event;

58. 137

59. 138     /* Log this kernel call. */

60. 139     TX_EL_BLOCK_ALLOCATE_INSERT

61. 140

62. 141     /* Store -1 in the third event slot. */

```

```

63. 142      *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_3_OFFSET)) = (ULONG) -1;

64. 143

65. 144      /* Save the time stamp for later comparison to verify that

66. 145          the event hasn't been overwritten by the time the allocate

67. 146          call succeeds. */

68. 147      lower_tbu = *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_LOWER_OFFSET));

69. 148      upper_tbu = *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_UPPER_OFFSET));

70. 149 #endif

71. 150

72. 151      /* Determine if there is an available block. */

73. 152      if (pool_ptr -> tx_block_pool_available != (UINT) 0) // 可用空闲内存块数量
        tx_block_pool_available不为0, 有可用空闲内存块, 那么获取内存块即可

74. 153      {

75. 154

76. 155          /* Yes, a block is available. Decrement the available count. */

77. 156          pool_ptr -> tx_block_pool_available--; // 可用空闲内存块数量减1

78. 157

79. 158          /* Pickup the current block pointer. */

80. 159          work_ptr = pool_ptr -> tx_block_pool_available_list; // 获取第一可用空
        闲内存块(所有内存大小都一样, 只需要获取可用空闲内存块链表表头的第一个内存块即可)

81. 160

82. 161          /* Return the first available block to the caller. */

83. 162          temp_ptr = TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR *))); //
        temp_ptr等于work_ptr加上一个指针(内存块的前面一个指针大小用与指向内存块所在的pool,
        ThreadX的内存块释放函数没有指定pool, 所以要在内存块的前面保存pool)

84. 163          return_ptr = TX_INDIRECT_VOID_TO_UCHAR_POINTER_CONVERT(block_ptr); //
        block_ptr强制转换为UCHAR **类型(类似malloc, 内存指针都是void *)

85. 164          *return_ptr = temp_ptr; // 申请到的内存地址

86. 165

87. 166          /* Modify the available list to point at the next block in the pool. */

88. 167          next_block_ptr = TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr);
        // 下一个可用空闲内存块

89. 168          pool_ptr -> tx_block_pool_available_list = *next_block_ptr; // pool的可
        用空闲内存块链表指向下一个空闲内存块

90. 169

```

```

91. 170      /* Save the pool's address in the block for when it is released! */
92. 171      temp_ptr = TX_BLOCK_POOL_TO_UCHAR_POINTER_CONVERT(pool_ptr); // pool的
地址
93. 172      *next_block_ptr = temp_ptr; // pool地址写入申请的内存块的前面(释放内存
块时，找到对应的pool，放回到对应的pool)
94. 173
95. 174 #ifdef TX_ENABLE_EVENT_TRACE
96. 175
97. 176      /* Check that the event time stamp is unchanged. A different
98. 177      timestamp means that a later event wrote over the byte
99. 178      allocate event. In that case, do nothing here. */
100. 179      if (entry_ptr != TX_NULL)
101. 180      {
102. 181
103. 182          /* Is the time stamp the same? */
104. 183          if (time_stamp == entry_ptr -> tx_trace_buffer_entry_time_stamp)
105. 184          {
106. 185
107. 186              /* Timestamp is the same, update the entry with the address. */
108. 187 #ifdef TX_MISRA_ENABLE
109. 188              entry_ptr -> tx_trace_buffer_entry_info_2 =
TX_POINTER_TO_ULONG_CONVERT(*block_ptr);
110. 189 #else
111. 190              entry_ptr -> tx_trace_buffer_entry_information_field_2 =
TX_POINTER_TO_ULONG_CONVERT(*block_ptr);
112. 191 #endif
113. 192          }
114. 193      }
115. 194 #endif
116. 195
117. 196 #ifdef TX_ENABLE_EVENT_LOGGING
118. 197      /* Store the address of the allocated block. */
119. 198      *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_3_OFFSET)) = (ULONG)
*block_ptr;

```

```

120. 199 #endif

121. 200

122. 201         /* Set status to success. */

123. 202         status = TX_SUCCESS;

124. 203

125. 204         /* Restore interrupts. */

126. 205         TX_RESTORE // 申请到了内存，开启中断

127. 206     }

128. 207     else // 没有可用空闲内存块的清空下

129. 208     {

130. 209

131. 210         /* Default the return pointer to NULL. */

132. 211         return_ptr = TX_INDIRECT_VOID_TO_UCHAR_POINTER_CONVERT(block_ptr);

133. 212         *return_ptr = TX_NULL; // 设置返回内存的地址为null

134. 213

135. 214         /* Determine if the request specifies suspension. */

136. 215         if (wait_option != TX_NO_WAIT) // 非TX_NO_WAIT，也就是申请内存的线程在没
            申请到内存时要等待有可用内存或者等待超时

137. 216         {

138. 217

139. 218             /* Determine if the preempt disable flag is non-zero. */

140. 219             if (_tx_thread_preempt_disable != ((UINT) 0)) // 如果禁用了抢占，那
            么线程不能阻塞，否则其他线程也得不到调度，暂用内存块的线程得不到调度内存就不可能释
            放，内核会进入死循环

141. 220             {

142. 221

143. 222                 /* Suspension is not allowed if the preempt disable flag is non-
                zero at this point, return error completion. */

144. 223                 status = TX_NO_MEMORY; // 申请不到内存又不能进入阻塞状态，设置
                没又内存可用，后面返回上一级函数

145. 224

146. 225                 /* Restore interrupts. */

147. 226                 TX_RESTORE

148. 227             }

```

```

149. 228         else // 允许抢占，线程可以进入阻塞状态
150. 229         {
151. 230
152. 231             /* Prepare for suspension of this thread. */
153. 232
154. 233 #ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO
155. 234
156. 235             /* Increment the total suspensions counter. */
157. 236             _tx_block_pool_performance_suspension_count++;
158. 237
159. 238             /* Increment the number of suspensions on this pool. */
160. 239             pool_ptr -> tx_block_pool_performance_suspension_count++;
161. 240 #endif
162. 241
163. 242             /* Pickup thread pointer. */
164. 243             TX_THREAD_GET_CURRENT(thread_ptr) // 获取当前线程指针
165. 244
166. 245             /* Setup cleanup routine pointer. */
167. 246             thread_ptr -> tx_thread_suspend_cleanup = &
            (_tx_block_pool_cleanup); // 设置挂起清理函数指针(阻塞被唤醒后执行的函数)
168. 247
169. 248             /* Setup cleanup information, i.e. this pool control
170. 249                 block. */
171. 250             thread_ptr -> tx_thread_suspend_control_block = (VOID *)
            pool_ptr; // pool信息
172. 251
173. 252             /* Save the return block pointer address as well. */
174. 253             thread_ptr -> tx_thread_additional_suspend_info = (VOID *)
            block_ptr; // 内存地址
175. 254
176. 255 #ifndef TX_NOT_INTERRUPTABLE
177. 256
178. 257             /* Increment the suspension sequence number, which is used to
            identify

```

```

179. 258             this suspension event. */
180. 259             thread_ptr -> tx_thread_suspension_sequence++;
181. 260 #endif
182. 261
183. 262             /* Pickup the number of suspended threads. */
184. 263             suspended_count = (pool_ptr -> tx_block_pool_suspended_count);
185. 264
186. 265             /* Increment the number of suspended threads. */
187. 266             (pool_ptr -> tx_block_pool_suspended_count)++; // 因等待内存而挂
起的线程数加1
188. 267
189. 268             /* Setup suspension list. */
190. 269             if (suspended_count == TX_NO_SUSPENSIONS) // 如果当前线程是第一
一个等待内存的线程，把当前线程挂载到pool的挂起链表tx_block_pool_suspension_list即可
191. 270             {
192. 271
193. 272                 /* No other threads are suspended. Setup the head pointer
and
194. 273                 just setup this threads pointers to itself. */
195. 274                 pool_ptr -> tx_block_pool_suspension_list =      thread_ptr;
196. 275                 thread_ptr -> tx_thread_suspended_next =      thread_ptr;
197. 276                 thread_ptr -> tx_thread_suspended_previous =    thread_ptr;
198. 277             }
199. 278             else // 还有其他线程也在等待内存块，当前线程添加到
tx_block_pool_suspension_list末尾即可
200. 279             {
201. 280
202. 281                 /* This list is not NULL, add current thread to the end. */
203. 282                 next_thread =                                     pool_ptr ->
tx_block_pool_suspension_list;
204. 283                 thread_ptr -> tx_thread_suspended_next =      next_thread;
205. 284                 previous_thread =                                next_thread
-> tx_thread_suspended_previous;
206. 285                 thread_ptr -> tx_thread_suspended_previous =
previous_thread;

```



```

207. 286             previous_thread -> tx_thread_suspended_next =  thread_ptr;
208. 287             next_thread -> tx_thread_suspended_previous =  thread_ptr;
209. 288         }
210. 289
211. 290             /* Set the state to suspended.  */
212. 291             thread_ptr -> tx_thread_state =  TX_BLOCK_MEMORY; // 设置线程阻塞状态(之前有介绍挂起线程操作，如果线程处于阻塞状态，那么需要延迟挂起，否则需要将线程从tx_block_pool_suspension_list删除，当有可用内存块的时候，可用内存就不会分配给不在等待链表里面的线程)
213. 292
214. 293 #ifdef TX_NOT_INTERRUPTABLE
215. 294
216. 295             /* Call actual non-interruptable thread suspension routine.  */
217. 296             _tx_thread_system_ni_suspend(thread_ptr, wait_option);
218. 297
219. 298             /* Restore interrupts.  */
220. 299             TX_RESTORE
221. 300 #else
222. 301
223. 302             /* Set the suspending flag.  */
224. 303             thread_ptr -> tx_thread_suspending =  TX_TRUE;
225. 304
226. 305             /* Setup the timeout period.  */
227. 306             thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks
=  wait_option; // 超时时间(_tx_thread_system_suspend会根据该值决定是否启动定时器，线程如果在超时时间内没有获取到内存，那么超时定时器将被调用，线程将被唤醒)
228. 307
229. 308             /* Temporarily disable preemption.  */
230. 309             _tx_thread_preempt_disable++;
231. 310
232. 311             /* Restore interrupts.  */
233. 312             TX_RESTORE
234. 313
235. 314             /* Call actual thread suspension routine.  */

```

```

236. 315             _tx_thread_system_suspend(thread_ptr); // 挂起当前线程

237. 316 #endif

238. 317

239. 318 #ifdef TX_ENABLE_EVENT_TRACE

240. 319

241. 320             /* Check that the event time stamp is unchanged. A different
242. 321                 timestamp means that a later event wrote over the byte
243. 322                 allocate event. In that case, do nothing here. */
244. 323             if (entry_ptr != TX_NULL)
245. 324             {
246. 325
247. 326                 /* Is the time-stamp the same? */
248. 327                 if (time_stamp == entry_ptr ->
249. 328                     tx_trace_buffer_entry_time_stamp)
250. 329
251. 330                     /* Timestamp is the same, update the entry with the
252. 331                         address. */
253. 332                     entry_ptr -> tx_trace_buffer_entry_info_2 =
254. 333                         TX_POINTER_TO_ULONG_CONVERT(*block_ptr);
255. 334
256. 335 #endif
257. 336             }
258. 337         }
259. 338 #endif
260. 339
261. 340 #ifdef TX_ENABLE_EVENT_LOGGING

262. 341             /* Check that the event time stamp is unchanged and the call is
263. 342                 about
                to return success. A different timestamp means that a later
                event

```

```

264. 343                wrote over the block allocate event.  A return value other
                than

265. 344                TX_SUCCESS indicates that no block was available. In those
                cases,

266. 345                do nothing here.  */

267. 346                if (lower_tbu == *((ULONG *) (log_entry_ptr +
                TX_EL_EVENT_TIME_LOWER_OFFSET)) &&

268. 347                upper_tbu == *((ULONG *) (log_entry_ptr +
                TX_EL_EVENT_TIME_UPPER_OFFSET)) &&

269. 348                ((thread_ptr -> tx_thread_suspend_status) == TX_SUCCESS))

270. 349                {

271. 350

272. 351                /* Store the address of the allocated block.  */

273. 352                *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_3_OFFSET)) =
                (ULONG) *block_ptr;

274. 353                }

275. 354 #endif

276. 355

277. 356                /* Return the completion status.  */

278. 357                status = thread_ptr -> tx_thread_suspend_status; // 线程挂起状
                态(释放内存块的线程会把内存块直接给等待内存的线程，也就是
                tx_thread_additional_suspend_info(block_ptr))

279. 358                }

280. 359                }

281. 360                else

282. 361                {

283. 362

284. 363                /* Immediate return, return error completion.  */

285. 364                status = TX_NO_MEMORY;

286. 365

287. 366                /* Restore interrupts.  */

288. 367                TX_RESTORE

289. 368                }

290. 369                }

291. 370

```

```
292. 371      /* Return completion status.  */  
293. 372      return(status);  
294. 373 }
```



## 1.3、申请内存块超时

---

线程申请内存块超时，主要是超时定时器处理，超时定时器处理线程

`_tx_timer_thread_entry`调用线程超时函数`_tx_thread_timeout`处理线程超时事件，`_tx_thread_timeout`调用线程超时的处理函数`tx_thread_suspend_cleanup`，对应等待内存的线程回调函数就指向`_tx_block_pool_cleanup`。

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

```

1. 078 VOID _tx_block_pool_cleanup(TX_THREAD *thread_ptr, ULONG suspension_sequence)

2. 079 {

3. 080

4. 081 #ifndef TX_NOT_INTERRUPTABLE

5. 082 TX_INTERRUPT_SAVE_AREA

6. 083 #endif

7. 084

8. 085 TX_BLOCK_POOL      *pool_ptr;

9. 086 UINT                suspended_count;

10. 087 TX_THREAD          *next_thread;

11. 088 TX_THREAD          *previous_thread;

12. 089

13. 090

14. 091 #ifndef TX_NOT_INTERRUPTABLE

15. 092

16. 093     /* Disable interrupts to remove the suspended thread from the block pool.
        */

17. 094     TX_DISABLE

18. 095

19. 096     /* Determine if the cleanup is still required. */

20. 097     if (thread_ptr -> tx_thread_suspend_cleanup == &(_tx_block_pool_cleanup)) //
        再次检查tx_thread_suspend_cleanup函数，有可能超时处理过程，释放内存的线程已经把内存
        分配给了超时的线程(超时和内存释放同时发生，保证不了谁先执行)

21. 098     {

22. 099

23. 100         /* Check for valid suspension sequence. */

24. 101         if (suspension_sequence == thread_ptr -> tx_thread_suspension_sequence)

25. 102         {

26. 103

27. 104             /* Setup pointer to block pool control block. */

28. 105             pool_ptr = TX_VOID_TO_BLOCK_POOL_POINTER_CONVERT(thread_ptr ->
                tx_thread_suspend_control_block); // 获取等待的pool

29. 106

```

```

30. 107          /* Check for a NULL byte pool pointer.  */
31. 108          if (pool_ptr != TX_NULL) // pool不为空(再次核对pool相关信息)
32. 109          {
33. 110
34. 111          /* Check for valid pool ID.  */
35. 112          if (pool_ptr -> tx_block_pool_id == TX_BLOCK_POOL_ID) // pool类
           型为块类型
36. 113          {
37. 114
38. 115          /* Determine if there are any thread suspensions.  */
39. 116          if (pool_ptr -> tx_block_pool_suspended_count !=
TX_NO_SUSPENSIONS) // 是否有等待内存的线程(如果没有的话，那么需要唤醒的线程应该就不
           再等待内存块了；ThreadX很多开关中断操作，有些关联操作不保证原子操作，所以有多重检查)
40. 117          {
41. 118 #else
42. 119
43. 120          /* Setup pointer to block pool control block.  */
44. 121          pool_ptr =
TX_VOID_TO_BLOCK_POOL_POINTER_CONVERT(thread_ptr ->
tx_thread_suspend_control_block); // 获取等待的pool
45. 122 #endif
46. 123
47. 124          /* Yes, we still have thread suspension!  */
48. 125
49. 126          /* Clear the suspension cleanup flag.  */
50. 127          thread_ptr -> tx_thread_suspend_cleanup = TX_NULL; //
           清空线程挂起清理函数指针
51. 128
52. 129          /* Decrement the suspended count.  */
53. 130          pool_ptr -> tx_block_pool_suspended_count--; // 线程超
           时，不再等待内存块，pool等待挂起计数器减1
54. 131
55. 132          /* Pickup the suspended count.  */
56. 133          suspended_count = pool_ptr ->
tx_block_pool_suspended_count;
57. 134

```

[illegible]





```

112. 189                                _tx_thread_system_ni_resume(thread_ptr);

113. 190 #else

114. 191                                /* Temporarily disable preemption. */

115. 192                                _tx_thread_preempt_disable++;

116. 193

117. 194                                /* Restore interrupts. */

118. 195                                TX_RESTORE

119. 196

120. 197                                /* Resume the thread! */

121. 198                                _tx_thread_system_resume(thread_ptr); // 唤醒阻塞的
    等待内存的线程

122. 199

123. 200                                /* Disable interrupts. */

124. 201                                TX_DISABLE

125. 202 #endif

126. 203                                }

127. 204 #ifndef TX_NOT_INTERRUPTABLE

128. 205                                }

129. 206                                }

130. 207                                }

131. 208                                }

132. 209                                }

133. 210

134. 211                                /* Restore interrupts. */

135. 212                                TX_RESTORE

136. 213 #endif

137. 214 }

```



## 1.4、block内存释放\_tx\_block\_release

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

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

`_tx_block_release`代码比较简单, 实现如下:

```

1. 075 UINT   _tx_block_release(VOID *block_ptr)

2. 076 {

3. 077

4. 078 TX_INTERRUPT_SAVE_AREA

5. 079

6. 080 TX_BLOCK_POOL      *pool_ptr;

7. 081 TX_THREAD          *thread_ptr;

8. 082 UCHAR              *work_ptr;

9. 083 UCHAR              **return_block_ptr;

10. 084 UCHAR             **next_block_ptr;

11. 085 UINT              suspended_count;

12. 086 TX_THREAD         *next_thread;

13. 087 TX_THREAD         *previous_thread;

14. 088

15. 089

16. 090      /* Disable interrupts to put this block back in the pool. */

17. 091      TX_DISABLE

18. 092

19. 093      /* Pickup the pool pointer which is just previous to the starting

20. 094         address of the block that the caller sees. */

21. 095      work_ptr =      TX_VOID_TO_UCHAR_POINTER_CONVERT(block_ptr);

22. 096      work_ptr =      TX_UCHAR_POINTER_SUB(work_ptr, (sizeof(UCHAR *)));

23. 097      next_block_ptr = TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr);

24. 098      pool_ptr =      TX_UCHAR_TO_BLOCK_POOL_POINTER_CONVERT((*next_block_ptr));

25. 099

26. 100 #ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO

27. 101

28. 102      /* Increment the total releases counter. */

29. 103      _tx_block_pool_performance_release_count++;

30. 104

31. 105      /* Increment the number of releases on this pool. */

```

```

32. 106     pool_ptr -> tx_block_pool_performance_release_count++;

33. 107 #endif

34. 108

35. 109     /* If trace is enabled, insert this event into the trace buffer. */

36. 110     TX_TRACE_IN_LINE_INSERT(TX_TRACE_BLOCK_RELEASE, pool_ptr,
    TX_POINTER_TO_ULONG_CONVERT(block_ptr), pool_ptr -> tx_block_pool_suspended_count,
    TX_POINTER_TO_ULONG_CONVERT(&work_ptr), TX_TRACE_BLOCK_POOL_EVENTS)

37. 111

38. 112     /* Log this kernel call. */

39. 113     TX_EL_BLOCK_RELEASE_INSERT

40. 114

41. 115     /* Determine if there are any threads suspended on the block pool. */

42. 116     thread_ptr = pool_ptr -> tx_block_pool_suspension_list;

43. 117     if (thread_ptr != TX_NULL)

44. 118     {

45. 119

46. 120         /* Remove the suspended thread from the list. */

47. 121

48. 122         /* Decrement the number of threads suspended. */

49. 123         (pool_ptr -> tx_block_pool_suspended_count)--;

50. 124

51. 125         /* Pickup the suspended count. */

52. 126         suspended_count = (pool_ptr -> tx_block_pool_suspended_count);

53. 127

54. 128         /* See if this is the only suspended thread on the list. */

55. 129         if (suspended_count == TX_NO_SUSPENSIONS)

56. 130         {

57. 131

58. 132             /* Yes, the only suspended thread. */

59. 133

60. 134             /* Update the head pointer. */

61. 135             pool_ptr -> tx_block_pool_suspension_list = TX_NULL;

62. 136     }

```

```

63. 137         else
64. 138         {
65. 139
66. 140             /* At least one more thread is on the same expiration list. */
67. 141
68. 142             /* Update the list head pointer. */
69. 143             next_thread =                                thread_ptr ->
tx_thread_suspended_next;
70. 144             pool_ptr -> tx_block_pool_suspension_list = next_thread;
71. 145
72. 146             /* Update the links of the adjacent threads. */
73. 147             previous_thread =                                thread_ptr ->
tx_thread_suspended_previous;
74. 148             next_thread -> tx_thread_suspended_previous = previous_thread;
75. 149             previous_thread -> tx_thread_suspended_next = next_thread;
76. 150         }
77. 151
78. 152         /* Prepare for resumption of the first thread. */
79. 153
80. 154         /* Clear cleanup routine to avoid timeout. */
81. 155         thread_ptr -> tx_thread_suspend_cleanup = TX_NULL;
82. 156
83. 157         /* Return this block pointer to the suspended thread waiting for
84. 158            a block. */
85. 159         return_block_ptr = TX_VOID_TO_INDIRECT_UCHAR_POINTER_CONVERT(thread_ptr
-> tx_thread_additional_suspend_info);
86. 160         work_ptr = TX_VOID_TO_UCHAR_POINTER_CONVERT(block_ptr);
87. 161         *return_block_ptr = work_ptr;
88. 162
89. 163         /* Put return status into the thread control block. */
90. 164         thread_ptr -> tx_thread_suspend_status = TX_SUCCESS;
91. 165
92. 166 #ifdef TX_NOT_INTERRUPTABLE

```

```

93. 167
94. 168     /* Resume the thread! */
95. 169     _tx_thread_system_ni_resume(thread_ptr);
96. 170
97. 171     /* Restore interrupts. */
98. 172     TX_RESTORE
99. 173 #else
100. 174
101. 175     /* Temporarily disable preemption. */
102. 176     _tx_thread_preempt_disable++;
103. 177
104. 178     /* Restore interrupts. */
105. 179     TX_RESTORE
106. 180
107. 181     /* Resume thread. */
108. 182     _tx_thread_system_resume(thread_ptr);
109. 183 #endif
110. 184     }
111. 185     else
112. 186     {
113. 187
114. 188         /* No thread is suspended for a memory block. */
115. 189
116. 190         /* Put the block back in the available list. */
117. 191         *next_block_ptr = pool_ptr -> tx_block_pool_available_list;
118. 192
119. 193         /* Adjust the head pointer. */
120. 194         pool_ptr -> tx_block_pool_available_list = work_ptr;
121. 195
122. 196         /* Increment the count of available blocks. */
123. 197         pool_ptr -> tx_block_pool_available++;
124. 198

```

```

125. 199          /* Restore interrupts. */
126. 200          TX_RESTORE
127. 201      }
128. 202
129. 203          /* Return successful completion status. */
130. 204          return(TX_SUCCESS);
131. 205 }

```

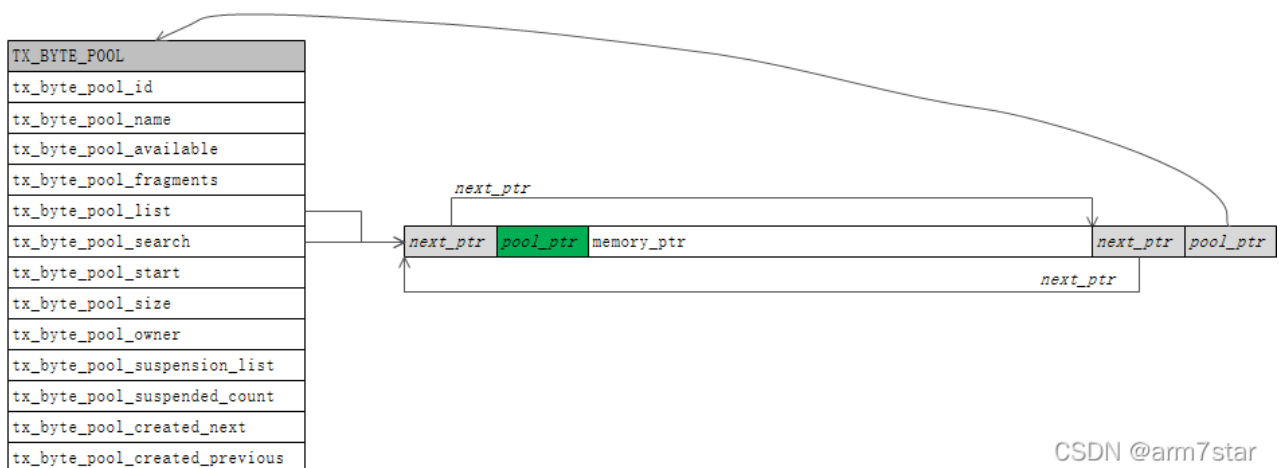


## 2、byte内存管理

### 2.1、byte内存管理介绍

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

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



CSDN @arm7star

### 2.2、byte内存申请\_tx\_byte\_allocate

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

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

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

\_tx\_byte\_allocate代码实现如下：



```

1. 082 UINT _tx_byte_allocate(TX_BYTE_POOL *pool_ptr, VOID **memory_ptr, ULONG
    memory_size, ULONG wait_option)

2. 083 {

3. 084

4. 085 TX_INTERRUPT_SAVE_AREA

5. 086

6. 087 UINT                                status;

7. 088 TX_THREAD                          *thread_ptr;

8. 089 UCHAR                              *work_ptr;

9. 090 UINT                                suspended_count;

10. 091 TX_THREAD                          *next_thread;

11. 092 TX_THREAD                          *previous_thread;

12. 093 UINT                                finished;

13. 094 #ifdef TX_ENABLE_EVENT_TRACE

14. 095 TX_TRACE_BUFFER_ENTRY               *entry_ptr;

15. 096 ULONG                                time_stamp = ((ULONG) 0);

16. 097 #endif

17. 098 #ifdef TX_ENABLE_EVENT_LOGGING

18. 099 UCHAR                                *log_entry_ptr;

19. 100 ULONG                                upper_tbu;

20. 101 ULONG                                lower_tbu;

21. 102 #endif

22. 103

23. 104

24. 105     /* Round the memory size up to the next size that is evenly divisible by

25. 106        an ALIGN_TYPE (this is typically a 32-bit ULONG). This guarantees proper
        alignment. */

26. 107     memory_size = (((memory_size + (sizeof(ALIGN_TYPE)))-((ALIGN_TYPE)
        1))/(sizeof(ALIGN_TYPE))) * (sizeof(ALIGN_TYPE));

27. 108

28. 109     /* Disable interrupts. */

29. 110     TX_DISABLE

30. 111

```

```
31. 112     /* Pickup thread pointer.  */
32. 113     TX_THREAD_GET_CURRENT(thread_ptr)
33. 114
34. 115 #ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO
35. 116
36. 117     /* Increment the total allocations counter.  */
37. 118     _tx_byte_pool_performance_allocate_count++;
38. 119
39. 120     /* Increment the number of allocations on this pool.  */
40. 121     pool_ptr -> tx_byte_pool_performance_allocate_count++;
41. 122 #endif
42. 123
43. 124 #ifdef TX_ENABLE_EVENT_TRACE
44. 125
45. 126     /* If trace is enabled, save the current event pointer.  */
46. 127     entry_ptr = _tx_trace_buffer_current_ptr;
47. 128
48. 129     /* If trace is enabled, insert this event into the trace buffer.  */
49. 130     TX_TRACE_IN_LINE_INSERT(TX_TRACE_BYTE_ALLOCATE, pool_ptr, 0, memory_size,
        wait_option, TX_TRACE_BYTE_POOL_EVENTS)
50. 131
51. 132     /* Save the time stamp for later comparison to verify that
52. 133        the event hasn't been overwritten by the time the allocate
53. 134        call succeeds.  */
54. 135     if (entry_ptr != TX_NULL)
55. 136     {
56. 137
57. 138         time_stamp = entry_ptr -> tx_trace_buffer_entry_time_stamp;
58. 139     }
59. 140 #endif
60. 141
61. 142 #ifdef TX_ENABLE_EVENT_LOGGING
```

```

62. 143     log_entry_ptr = *(UCHAR **) _tx_el_current_event;

63. 144

64. 145     /* Log this kernel call. */

65. 146     TX_EL_BYTE_ALLOCATE_INSERT

66. 147

67. 148     /* Store -1 in the fourth event slot. */

68. 149     *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_4_OFFSET)) = (ULONG) -1;

69. 150

70. 151     /* Save the time stamp for later comparison to verify that

71. 152         the event hasn't been overwritten by the time the allocate

72. 153         call succeeds. */

73. 154     lower_tbu = *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_LOWER_OFFSET));

74. 155     upper_tbu = *((ULONG *) (log_entry_ptr + TX_EL_EVENT_TIME_UPPER_OFFSET));

75. 156 #endif

76. 157

77. 158     /* Set the search finished flag to false. */

78. 159     finished = TX_FALSE;

79. 160

80. 161     /* Loop to handle cases where the owner of the pool changed. */

81. 162     do // 与block内存分配不同的是，block内存大小固定，释放内存时，肯定可以把释放
        的内存给另外一个等待内存的线程，但是byte内存释放时，释放的大小并不一定满足等待内存线
        程需要的内存的大小，因此有更多内存的时候，需要等待内存的线程自己去检查是否有足够空闲
        内存，所以这里是不断检查是否有足够空闲内存

82. 163     {

83. 164

84. 165         /* Indicate that this thread is the current owner. */

85. 166         pool_ptr -> tx_byte_pool_owner = thread_ptr; // 记录当前线程在操作该
        pool(主要是用于检查是否有别的线程操作了pool，如果没有找到合适大小的可用空闲内存块，
        在进入阻塞前，如果有其他线程操作了pool，那么可能pool里面可能存在可用空闲内存块，如果
        不再去检查的话也没有其他线程释放内存的话，当前线程可能就永远不会被唤醒了，只有内存释
        放的时候会去检查唤醒等待内存的线程)

86. 167

87. 168         /* Restore interrupts. */

88. 169         TX_RESTORE // 允许中断(byte内存申请使用频繁，耗时比较长，不能简单的互斥
        访问，否则并发度很低，影响性能)

89. 170

```

```

90. 171      /* At this point, the executing thread owns the pool and can perform a
      search

91. 172      for free memory.  */

92. 173      work_ptr = _tx_byte_pool_search(pool_ptr, memory_size); // 查找
      memory_size大小的空闲内存(_tx_byte_pool_search的时候会合并拆分空闲链表)

93. 174

94. 175      /* Optional processing extension.  */

95. 176      TX_BYTE_ALLOCATE_EXTENSION

96. 177

97. 178      /* Lockout interrupts.  */

98. 179      TX_DISABLE

99. 180

100. 181     /* Determine if we are finished.  */

101. 182     if (work_ptr != TX_NULL) // 如果找到了可用空闲内存，那么查找结束，
      work_ptr就是申请到的内存的地址

102. 183     {

103. 184

104. 185         /* Yes, we have found a block the search is finished.  */

105. 186         finished = TX_TRUE;

106. 187     }

107. 188     else // 没有找到可用的内存

108. 189     {

109. 190

110. 191         /* No block was found, does this thread still own the pool?  */

111. 192         if (pool_ptr -> tx_byte_pool_owner == thread_ptr) // 有别的线程修改
      了tx_byte_pool_owner，对pool进行了操作，如果别的线程释放了内存，那么要再次检查是否有
      可用内存(没有找到合适内存后会打开中断，如果开中断后有高优先级新城就绪或者其他情况导
      致有内存被释放，那么这里需要重新查找可用内存)，如果pool没有被修改，那么就要进入睡
      眠，此时中断已经关闭，不会有线程修改pool了

112. 193         {

113. 194

114. 195             /* Yes, then we have looked through the entire pool and haven't
      found the memory.  */

115. 196             finished = TX_TRUE; // 没有其他线程修改pool，不用再检查释放有可
      用内存

116. 197         }

```

```

117. 198         }

118. 199

119. 200     } while (finished == TX_FALSE); // finished为TX_FALSE, 表明没有获取到内存并
        且pool被修改了, 那么再次检查是否有可用内存

120. 201

121. 202     /* Copy the pointer into the return destination. */

122. 203     *memory_ptr = (VOID *) work_ptr; // 申请到的内存或者null

123. 204

124. 205     /* Determine if memory was found. */

125. 206     if (work_ptr != TX_NULL) // 申请到了内存, 返回成功, if里面很多代码都是性能统
        计或者其他trace的代码, 不用过多查看

126. 207     {

127. 208

128. 209 #ifdef TX_ENABLE_EVENT_TRACE

129. 210

130. 211         /* Check that the event time stamp is unchanged. A different

131. 212             timestamp means that a later event wrote over the byte

132. 213             allocate event. In that case, do nothing here. */

133. 214         if (entry_ptr != TX_NULL)

134. 215         {

135. 216

136. 217             /* Is the timestamp the same? */

137. 218             if (time_stamp == entry_ptr -> tx_trace_buffer_entry_time_stamp)

138. 219             {

139. 220

140. 221                 /* Timestamp is the same, update the entry with the address. */

141. 222 #ifdef TX_MISRA_ENABLE

142. 223                 entry_ptr -> tx_trace_buffer_entry_info_2 =
                    TX_POINTER_TO_ULONG_CONVERT(*memory_ptr);

143. 224 #else

144. 225                 entry_ptr -> tx_trace_buffer_entry_information_field_2 =
                    TX_POINTER_TO_ULONG_CONVERT(*memory_ptr);

145. 226 #endif

146. 227     }

```

```

147. 228     }

148. 229 #endif

149. 230

150. 231 #ifdef TX_ENABLE_EVENT_LOGGING

151. 232     /* Check that the event time stamp is unchanged.  A different
152. 233         timestamp means that a later event wrote over the byte
153. 234         allocate event.  In that case, do nothing here.  */

154. 235     if (lower_tbu == *((ULONG *) (log_entry_ptr +
    TX_EL_EVENT_TIME_LOWER_OFFSET)) &&

155. 236         upper_tbu == *((ULONG *) (log_entry_ptr +
    TX_EL_EVENT_TIME_UPPER_OFFSET)))

156. 237     {

157. 238         /* Store the address of the allocated fragment.  */

158. 239         *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_4_OFFSET)) = (ULONG)
    *memory_ptr;

159. 240     }

160. 241 #endif

161. 242

162. 243     /* Restore interrupts.  */

163. 244     TX_RESTORE

164. 245

165. 246     /* Set the status to success.  */

166. 247     status = TX_SUCCESS;

167. 248 }

168. 249 else // 没有找到可用内存，检查是否等待内存还是直接返回失败

169. 250 {

170. 251

171. 252     /* No memory of sufficient size was found...  */

172. 253

173. 254     /* Determine if the request specifies suspension.  */

174. 255     if (wait_option != TX_NO_WAIT) // 有等待选项

175. 256     {

176. 257

```

```

177. 258          /* Determine if the preempt disable flag is non-zero. */
178. 259          if (_tx_thread_preempt_disable != ((UINT) 0)) // 检查是否禁止抢占，
            如果禁止抢占的话，线程不能阻塞，否则会禁止其他线程的调度，其他就绪线程都没办法执行了
179. 260          {
180. 261
181. 262          /* Suspension is not allowed if the preempt disable flag is non-
            zero at this point - return error completion. */
182. 263          status = TX_NO_MEMORY; // 禁止抢占，不能阻塞，返回申请内存失败
            即可
183. 264
184. 265          /* Restore interrupts. */
185. 266          TX_RESTORE
186. 267          }
187. 268          else // 阻塞等待内存，超时处理过程与block内存超时一样，别的线程释放
            内存时，如果有足够内存的话，也是由释放内存的线程把内存给等待内存的线程(合并/拆分空闲
            内存块等)
188. 269          {
189. 270
190. 271          /* Prepare for suspension of this thread. */
191. 272
192. 273 #ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO
193. 274
194. 275          /* Increment the total suspensions counter. */
195. 276          _tx_byte_pool_performance_suspension_count++;
196. 277
197. 278          /* Increment the number of suspensions on this pool. */
198. 279          pool_ptr -> tx_byte_pool_performance_suspension_count++;
199. 280 #endif
200. 281
201. 282          /* Setup cleanup routine pointer. */
202. 283          thread_ptr -> tx_thread_suspend_cleanup = &
            (_tx_byte_pool_cleanup);
203. 284
204. 285          /* Setup cleanup information, i.e. this pool control
205. 286          block. */

```

```

206. 287         thread_ptr -> tx_thread_suspend_control_block = (VOID *)
        pool_ptr;

207. 288

208. 289         /* Save the return memory pointer address as well. */

209. 290         thread_ptr -> tx_thread_additional_suspend_info = (VOID *)
        memory_ptr; // 别的线程释放内存时，如果有可用内存分配给阻塞的线程，那么会把申请到的
        内存赋值给memory_ptr(被释放的内存可能满足内存大小，或者被释放内存合并相邻空闲块后的
        空闲内存块可能满足内存大小，所以释放内存时检查是否有合适内存效率应该高一些)

210. 291

211. 292         /* Save the byte size requested. */

212. 293         thread_ptr -> tx_thread_suspend_info = memory_size; // 需要申请
        的内存大小

213. 294

214. 295 #ifndef TX_NOT_INTERRUPTABLE

215. 296

216. 297         /* Increment the suspension sequence number, which is used to
        identify

217. 298         this suspension event. */

218. 299         thread_ptr -> tx_thread_suspension_sequence++;

219. 300 #endif

220. 301

221. 302         /* Pickup the number of suspended threads. */

222. 303         suspended_count = pool_ptr -> tx_byte_pool_suspended_count; //
        后续代码跟block阻塞挂起时原理一样

223. 304

224. 305         /* Increment the suspension count. */

225. 306         (pool_ptr -> tx_byte_pool_suspended_count)++;

226. 307

227. 308         /* Setup suspension list. */

228. 309         if (suspended_count == TX_NO_SUSPENSIONS)

229. 310         {

230. 311

231. 312         /* No other threads are suspended. Setup the head pointer
        and

232. 313         just setup this threads pointers to itself. */

233. 314         pool_ptr -> tx_byte_pool_suspension_list = thread_ptr;

```



```

234. 315             thread_ptr -> tx_thread_suspended_next =      thread_ptr;
235. 316             thread_ptr -> tx_thread_suspended_previous =    thread_ptr;
236. 317         }
237. 318     else
238. 319     {
239. 320
240. 321         /* This list is not NULL, add current thread to the end. */
241. 322         next_thread =                                           pool_ptr ->
tx_byte_pool_suspension_list;
242. 323         thread_ptr -> tx_thread_suspended_next =              next_thread;
243. 324         previous_thread =                                       next_thread
-> tx_thread_suspended_previous;
244. 325         thread_ptr -> tx_thread_suspended_previous =
previous_thread;
245. 326         previous_thread -> tx_thread_suspended_next =          thread_ptr;
246. 327         next_thread -> tx_thread_suspended_previous =          thread_ptr;
247. 328     }
248. 329
249. 330     /* Set the state to suspended. */
250. 331     thread_ptr -> tx_thread_state =          TX_BYTE_MEMORY;
251. 332
252. 333 #ifdef TX_NOT_INTERRUPTABLE
253. 334
254. 335     /* Call actual non-interruptable thread suspension routine. */
255. 336     _tx_thread_system_ni_suspend(thread_ptr, wait_option);
256. 337
257. 338     /* Restore interrupts. */
258. 339     TX_RESTORE
259. 340 #else
260. 341
261. 342     /* Set the suspending flag. */
262. 343     thread_ptr -> tx_thread_suspending =    TX_TRUE;
263. 344

```

```

264. 345          /* Setup the timeout period.  */
265. 346          thread_ptr -> tx_thread_timer.tx_timer_internal_remaining_ticks
           = wait_option;
266. 347
267. 348          /* Temporarily disable preemption.  */
268. 349          _tx_thread_preempt_disable++;
269. 350
270. 351          /* Restore interrupts.  */
271. 352          TX_RESTORE
272. 353
273. 354          /* Call actual thread suspension routine.  */
274. 355          _tx_thread_system_suspend(thread_ptr); // 挂起当前线程
275. 356 #endif
276. 357
277. 358 #ifdef TX_ENABLE_EVENT_TRACE
278. 359
279. 360          /* Check that the event time stamp is unchanged.  A different
280. 361             timestamp means that a later event wrote over the byte
281. 362             allocate event.  In that case, do nothing here.  */
282. 363          if (entry_ptr != TX_NULL)
283. 364          {
284. 365
285. 366              /* Is the timestamp the same?  */
286. 367              if (time_stamp == entry_ptr ->
                tx_trace_buffer_entry_time_stamp)
287. 368              {
288. 369
289. 370                  /* Timestamp is the same, update the entry with the
                address.  */
290. 371 #ifdef TX_MISRA_ENABLE
291. 372                  entry_ptr -> tx_trace_buffer_entry_info_2 =
                TX_POINTER_TO_ULONG_CONVERT(*memory_ptr);
292. 373 #else

```

```

293. 374                entry_ptr -> tx_trace_buffer_entry_information_field_2 =
                TX_POINTER_TO_ULONG_CONVERT(*memory_ptr);

294. 375 #endif

295. 376                }

296. 377                }

297. 378 #endif

298. 379

299. 380 #ifdef TX_ENABLE_EVENT_LOGGING

300. 381                /* Check that the event time stamp is unchanged. A different
301. 382                timestamp means that a later event wrote over the byte
302. 383                allocate event. In that case, do nothing here. */

303. 384                if (lower_tbu == *((ULONG *) (log_entry_ptr +
                TX_EL_EVENT_TIME_LOWER_OFFSET)) &&

304. 385                upper_tbu == *((ULONG *) (log_entry_ptr +
                TX_EL_EVENT_TIME_UPPER_OFFSET)))

305. 386                {

306. 387

307. 388                /* Store the address of the allocated fragment. */

308. 389                *((ULONG *) (log_entry_ptr + TX_EL_EVENT_INFO_4_OFFSET)) =
                (ULONG) *memory_ptr;

309. 390                }

310. 391 #endif

311. 392

312. 393                /* Return the completion status. */

313. 394                status = thread_ptr -> tx_thread_suspend_status; // 设置返回状
                态

314. 395                }

315. 396                }

316. 397                else

317. 398                {

318. 399

319. 400                /* Restore interrupts. */

320. 401                TX_RESTORE

321. 402

```

```

322. 403             /* Immediate return, return error completion.  */
323. 404             status = TX_NO_MEMORY;
324. 405         }
325. 406     }
326. 407
327. 408     /* Return completion status.  */
328. 409     return(status);
329. 410 }

```



## 2.3、byte空闲内存块查找\_tx\_byte\_pool\_search

---

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

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

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

\_tx\_byte\_pool\_search实现代码如下：

```

1. 084 UCHAR   *_tx_byte_pool_search(TX_BYTE_POOL *pool_ptr, ULONG memory_size)

2. 085 {

3. 086

4. 087 TX_INTERRUPT_SAVE_AREA

5. 088

6. 089 UCHAR           *current_ptr;

7. 090 UCHAR           *next_ptr;

8. 091 UCHAR           **this_block_link_ptr;

9. 092 UCHAR           **next_block_link_ptr;

10. 093 ULONG           available_bytes;

11. 094 UINT            examine_blocks;

12. 095 UINT            first_free_block_found = TX_FALSE;

13. 096 TX_THREAD        *thread_ptr;

14. 097 ALIGN_TYPE       *free_ptr;

15. 098 UCHAR           *work_ptr;

16. 099

17. 100

18. 101     /* Disable interrupts. */

19. 102     TX_DISABLE

20. 103

21. 104     /* First, determine if there are enough bytes in the pool. */

22. 105     if (memory_size >= pool_ptr->tx_byte_pool_available) // 申请的内存大于等于
        所有可以用的内存大小(还要预留部分内存控制块，所以内存不够，返回null)

23. 106     {

24. 107

25. 108         /* Restore interrupts. */

26. 109         TX_RESTORE

27. 110

28. 111         /* Not enough memory, return a NULL pointer. */

29. 112         current_ptr = TX_NULL;

30. 113     }

```

```

31. 114     else // 总的可用内存满足申请的内存大小(可能存在满足大小的连续内存块,需要查
           找一下)

32. 115     {

33. 116

34. 117         /* Pickup thread pointer. */

35. 118         TX_THREAD_GET_CURRENT(thread_ptr)

36. 119

37. 120         /* Setup ownership of the byte pool. */

38. 121         pool_ptr -> tx_byte_pool_owner = thread_ptr; // 记录正在操作pool的线
           程,后面会开中断,如果开中断后,没有线程对pool操作,也就是pool没有被改变,那么可以接
           着之前的查找继续,否则得重新查找(也就是在多线程同时申请内存的时候效率比较低,查找空
           闲内存也不能阻塞高优先级线程申请内存,如果有高优先级线程申请内存,那么高优先级线程应
           该先获取内存)

39. 122

40. 123         /* Walk through the memory pool in search for a large enough block. */

41. 124         current_ptr = pool_ptr -> tx_byte_pool_search; // 从
           tx_byte_pool_search开始查找空闲内存

42. 125         examine_blocks = pool_ptr -> tx_byte_pool_fragments + ((UINT) 1); //
           tx_byte_pool_fragments内存碎片数量,也就是有多少个内存块(加1???多检查一次也不影响,
           暂时不考虑什么情况存有多检查一次的必要)

43. 126         available_bytes = ((ULONG) 0);

44. 127         do

45. 128         {

46. 129

47. 130

48. 131 #ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

49. 132

50. 133             /* Increment the total fragment search counter. */

51. 134             _tx_byte_pool_performance_search_count++;

52. 135

53. 136             /* Increment the number of fragments searched on this pool. */

54. 137             pool_ptr -> tx_byte_pool_performance_search_count++;

55. 138 #endif

56. 139

57. 140             /* Check to see if this block is free. */

```

```

58. 141         work_ptr = TX_UCHAR_POINTER_ADD(current_ptr, (sizeof(UCHAR *))); //
    内存块加上UCHAR *大小的地址(current_ptr最前面保存的是下一个内存块的地址)

59. 142         free_ptr = TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(work_ptr); //
    work_ptr指针转换为ALIGN_TYPE类型指针

60. 143         if ((*free_ptr) == TX_BYTE_BLOCK_FREE) // 检查是否是空闲内存

61. 144         {

62. 145

63. 146             /* Determine if this is the first free block. */

64. 147             if (first_free_block_found == TX_FALSE) // 没有找到过空闲块时，
    first_free_block_found为TX_FALSE，如果找到了空闲块，first_free_block_found为
    TX_TRUE(内核要记录第一个找到的空闲块，避免每次都从非空闲块开始查找空闲内存)

65. 148         {

66. 149

67. 150             /* This is the first free block. */

68. 151             pool_ptr->tx_byte_pool_search = current_ptr; // 记录第一次
    找到的空闲内存块

69. 152

70. 153             /* Set the flag to indicate we have found the first free

71. 154             block. */

72. 155             first_free_block_found = TX_TRUE; // 已经找到了第一个空闲内
    存块

73. 156         }

74. 157

75. 158             /* Block is free, see if it is large enough. */

76. 159

77. 160             /* Pickup the next block's pointer. */

78. 161             this_block_link_ptr =
    TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(current_ptr); // 空闲内存块的最前面记录的
    是下一个相邻内存块的地址

79. 162             next_ptr = *this_block_link_ptr; // 获取下一个内存块
    的地址

80. 163

81. 164             /* Calculate the number of bytes available in this block. */

82. 165             available_bytes = TX_UCHAR_POINTER_DIF(next_ptr, current_ptr);
    // 下一个内存块的地址减去当前内存块的地址就是当前内存块的大小

83. 166             available_bytes = available_bytes - ((sizeof(UCHAR *)) +
    (sizeof(ALIGN_TYPE))); // 当前内存块大小减去一个指针以及块空闲类型的标志就是当前内存
    块可以返回给应用程序的可用内存大小

```

```

84. 167
85. 168          /* If this is large enough, we are done because our first-fit
    algorithm
86. 169          has been satisfied!  */
87. 170          if (available_bytes >= memory_size) // 可用内存大小大于等于申请的
    内存，那么从当前内存块申请内存给应用程序即可
88. 171          {
89. 172          /* Get out of the search loop!  */
90. 173          break;
91. 174          }
92. 175          else // 内存块可用内存大小不够，检查下一个相邻的内存块是否空闲，
    是否可以合并
93. 176          {
94. 177
95. 178          /* Clear the available bytes variable.  */
96. 179          available_bytes = ((ULONG) 0);
97. 180
98. 181          /* Not enough memory, check to see if the neighbor is
99. 182          free and can be merged.  */
100. 183          work_ptr = TX_UCHAR_POINTER_ADD(next_ptr, (sizeof(UCHAR
    *))); // 获取下一个相邻的内存块
101. 184          free_ptr =
    TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(work_ptr); // 内存块空闲类型
102. 185          if ((*free_ptr) == TX_BYTE_BLOCK_FREE) // 内存块空闲
103. 186          {
104. 187
105. 188          /* Yes, neighbor block can be merged! This is quickly
    accomplished
106. 189          by updating the current block with the next blocks
    pointer.  */
107. 190          next_block_link_ptr =
    TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(next_ptr); // 获取再下一个内存块地址(两个
    内存块地址相减才能知道内存块的大小，内存控制块没有记录内存大小，只记录相邻的下一个
    内存块的地址)
108. 191          *this_block_link_ptr = *next_block_link_ptr; // 当前内
    存块current_ptr的下一个内存块地址指向下一个内存块next_ptr的下一个内存块(next_ptr内存
    被合并了)
109. 192

```



```

110. 193                                     /* Reduce the fragment total.  We don't need to increase
      the bytes

111. 194                                     available because all free headers are also included
      in the available

112. 195                                     count.  */

113. 196                                     pool_ptr -> tx_byte_pool_fragments--; // 当前内存块与相
      邻的下一个内存块合并，内存碎片数减1

114. 197

115. 198 #ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

116. 199

117. 200                                     /* Increment the total merge counter.  */

118. 201                                     _tx_byte_pool_performance_merge_count++;

119. 202

120. 203                                     /* Increment the number of blocks merged on this pool.
      */

121. 204                                     pool_ptr -> tx_byte_pool_performance_merge_count++;

122. 205 #endif

123. 206

124. 207                                     /* See if the search pointer is affected.  */

125. 208                                     if (pool_ptr -> tx_byte_pool_search == next_ptr) // 如
      果tx_byte_pool_search指向被合并的下一个空闲内存块，那么更新tx_byte_pool_search指向合
      并后的空闲内存块(旧的内存块不存在了)

126. 209                                     {

127. 210

128. 211                                     /* Yes, update the search pointer.  */

129. 212                                     pool_ptr -> tx_byte_pool_search = current_ptr;

130. 213                                     }

131. 214                                     }

132. 215                                     else // 下一个内存块不空闲

133. 216                                     {

134. 217

135. 218                                     /* Neighbor is not free so we can skip over it!  */

136. 219                                     next_block_link_ptr =
      TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(next_ptr);

137. 220                                     current_ptr = *next_block_link_ptr; // current_ptr指向
      下下一个内存块，下一个非空闲内存块

```

```

138. 221
139. 222                                     /* Decrement the examined block count to account for
      this one. */
140. 223                                     if (examine_blocks != ((UINT) 0)) // examine_blocks不为
      0, 要检查的内存块数量减1(next_ptr检查过了, current_ptr的examine_blocks减1在后面)
141. 224                                     {
142. 225
143. 226                                     examine_blocks--;
144. 227
145. 228 #ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO
146. 229
147. 230                                     /* Increment the total fragment search counter. */
148. 231                                     _tx_byte_pool_performance_search_count++;
149. 232
150. 233                                     /* Increment the number of fragments searched on
      this pool. */
151. 234                                     pool_ptr -> tx_byte_pool_performance_search_count++;
152. 235 #endif
153. 236                                     }
154. 237                                     }
155. 238                                     }
156. 239                                     }
157. 240                                     else // 当前内存块不是空闲内存块, 更新current_ptr指向下一个内存块,
      检查下一个内存块
158. 241                                     {
159. 242
160. 243                                     /* Block is not free, move to next block. */
161. 244                                     this_block_link_ptr =
      TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(current_ptr);
162. 245                                     current_ptr = *this_block_link_ptr;
163. 246                                     }
164. 247
165. 248                                     /* Another block has been searched... decrement counter. */
166. 249                                     if (examine_blocks != ((UINT) 0)) // current_ptr检查过了,
      examine_blocks减1

```

```

167. 250      {
168. 251
169. 252          examine_blocks--;
170. 253      }
171. 254
172. 255      /* Restore interrupts temporarily. */
173. 256      TX_RESTORE // 允许中断，这个检查空闲内存块耗费了一定时间，不能阻塞中
    断
174. 257
175. 258      /* Disable interrupts. */
176. 259      TX_DISABLE // 再次关闭中断
177. 260
178. 261      /* Determine if anything has changed in terms of pool ownership. */
179. 262      if (pool_ptr -> tx_byte_pool_owner != thread_ptr) // 在检查下一个内
    存块前，检查一下pool的owner是不是当前线程，如果不是当前线程，那么pool的内存可能就被
    其他线程修改了，current_ptr可能被合并无效了，要检查的内存碎片数量有不一样了，所以又
    得重新查找空闲内存块
180. 263      {
181. 264
182. 265          /* Pool changed ownership in the brief period interrupts were
183. 266              enabled. Reset the search. */
184. 267              current_ptr =      pool_ptr -> tx_byte_pool_search; // 重新获取
    空闲内存查找的起始内存块(tx_byte_pool_search一般指向一个空闲内存块，内存申请的时候，
    如果该内存块被本次申请了，那么tx_byte_pool_search就指向下一个内存块(不一定是空闲
    的)；内存释放的时候，如果释放的内存块地址小于tx_byte_pool_search，那么更新
    tx_byte_pool_search指向刚释放的内存块地址，也就是ThreadX内核尽量从内存块链表前面的内
    存块分配内存，这样就会优先拆分前面的空闲内存块，后面的内存块可能就比较大小，尽量避免过
    多非连续的小内存碎片的产生)
185. 268              examine_blocks =      pool_ptr -> tx_byte_pool_fragments + ((UINT)
    1);
186. 269
187. 270          /* Setup our ownership again. */
188. 271          pool_ptr -> tx_byte_pool_owner =      thread_ptr;
189. 272      }
190. 273      } while(examine_blocks != ((UINT) 0)); // 如果所有内存块都检查完了还没找到
    到可用空闲内存块，那么退出循环
191. 274
192. 275      /* Determine if a block was found. If so, determine if it needs to be

```

```

193. 276         split.  */

194. 277         if (available_bytes != ((ULONG) 0)) // available_bytes不为0，表示找到了
            一个满足申请大小的内存块

195. 278     {

196. 279

197. 280         /* Determine if we need to split this block.  */

198. 281         if ((available_bytes - memory_size) >= ((ULONG) TX_BYTE_BLOCK_MIN))
            // 可用内存减去申请的内存还大于等于TX_BYTE_BLOCK_MIN，也就是将当前内存块拆分后，剩余的
            内存还够一个空闲内存块(内核不拆分成太小的内存块，太小的内存块分配管理起来效率太
            低，干脆就把多一点的内存块给应用程序即可，应用程序也不会访问多给的内存)

199. 282     {

200. 283

201. 284         /* Split the block.  */ // 拆分当前内存块

202. 285         next_ptr = TX_UCHAR_POINTER_ADD(current_ptr, (memory_size +
            ((sizeof(UCHAR *)) + (sizeof(ALIGN_TYPE))))); // "下一个内存块地址指针、内存块释放空
            闲字段 + 申请的内存大小"这么多作为一个新的内存块，里面的可用内存返回给应用程序；接下
            来的内存划分成一个新的空闲内存块，next_ptr即指向新的空闲内存块的地址

203. 286

204. 287         /* Setup the new free block.  */

205. 288         next_block_link_ptr =
            TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(next_ptr); // 指针转换，没有改变值

206. 289         this_block_link_ptr =
            TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(current_ptr); // 指针转换，没有改变值
            (current_ptr拆分前的下一个内存块地址，也就是拆分后的next_ptr内存块的下一个内存块地
            址)

207. 290         *next_block_link_ptr = *this_block_link_ptr; // next_ptr的下一
            个内存块地址，指向拆分前的current_ptr的下一个内存块地址

208. 291         work_ptr = TX_UCHAR_POINTER_ADD(next_ptr,
            (sizeof(UCHAR *))); // next_ptr + sizeof(UCHAR *)指向标记内存块是否空闲的地址

209. 292         free_ptr =
            TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(work_ptr);

210. 293         *free_ptr = TX_BYTE_BLOCK_FREE; // next_ptr设置为空
            闲

211. 294

212. 295         /* Increase the total fragment counter.  */

213. 296         pool_ptr -> tx_byte_pool_fragments++;

214. 297

215. 298         /* Update the current pointer to point at the newly created
            block.  */

```

```

216. 299             *this_block_link_ptr = next_ptr; // current_ptr的下一个内存块地
                址指向拆分出来的next_ptr内存块地址

217. 300

218. 301             /* Set available equal to memory size for subsequent
                calculation. */

219. 302             available_bytes = memory_size; // available_bytes等于拆分后申请
                到的内存块的可用内存大小

220. 303

221. 304 #ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

222. 305

223. 306             /* Increment the total split counter. */

224. 307             _tx_byte_pool_performance_split_count++;

225. 308

226. 309             /* Increment the number of blocks split on this pool. */

227. 310             pool_ptr -> tx_byte_pool_performance_split_count++;

228. 311 #endif

229. 312         }

230. 313

231. 314             /* In any case, mark the current block as allocated. */

232. 315             work_ptr = TX_UCHAR_POINTER_ADD(current_ptr,
                (sizeof(UCHAR *)));

233. 316             this_block_link_ptr =
                TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr);

234. 317             *this_block_link_ptr =
                TX_BYTE_POOL_TO_UCHAR_POINTER_CONVERT(pool_ptr); // 该地址指向pool地址(与标记内存释
                放空闲的标志占用同一个内存，pool_ptr不会等于空闲内存块标记，所以可以存pool指针，否则
                可能造成混乱；复用内存，这样就节省了一个内存空间)

235. 318

236. 319             /* Reduce the number of available bytes in the pool. */

237. 320             pool_ptr -> tx_byte_pool_available = (pool_ptr ->
                tx_byte_pool_available - available_bytes) - ((sizeof(UCHAR *)) +
                (sizeof(ALIGN_TYPE))); // pool的可用内存空间减去申请出去的内存块大小

238. 321

239. 322             /* Determine if the search pointer needs to be updated. This is only
                done

240. 323             if the search pointer matches the block to be returned. */

```

```

241. 324         if (current_ptr == pool_ptr -> tx_byte_pool_search) // 更新
tx_byte_pool_search指向当前内存块的下一个内存块(从tx_byte_pool_search开始查找空闲内存), tx_byte_pool_search指向的内存块已经被申请了, 下次不要再从这里开始查找, 免得浪费时间(更新的时候不保证下一个内存块可用, 所以前面的first_free_block_found也就是在查找过程中更新tx_byte_pool_search指向第一个空闲内存块)

242. 325         {

243. 326

244. 327             /* Yes, update the search pointer to the next block. */

245. 328             this_block_link_ptr =
TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(current_ptr);

246. 329             pool_ptr -> tx_byte_pool_search = *this_block_link_ptr; // 更新
tx_byte_pool_search, tx_byte_pool_search指向的内存块已经被申请了, 直接指向下一个内存块即可, 不管下一个内存块是否空闲, 由下次申请内存的线程来更新tx_byte_pool_search

247. 330         }

248. 331

249. 332         /* Restore interrupts. */

250. 333         TX_RESTORE

251. 334

252. 335         /* Adjust the pointer for the application. */

253. 336         current_ptr = TX_UCHAR_POINTER_ADD(current_ptr, (((sizeof(UCHAR *))
+ (sizeof(ALIGN_TYPE))))); // 修正current_ptr, 修正前current_ptr包含了pool指针、下一个内存块的地址, 跳过这些才是返回给应用程序的可用内存地址, 也就是c语言malloc返回的地址

254. 337     }

255. 338     else // 查找完了所以内存块也没找到合适的空闲内存, 返回空指针给上一级函数

256. 339     {

257. 340

258. 341         /* Restore interrupts. */

259. 342         TX_RESTORE

260. 343

261. 344         /* Set current pointer to NULL to indicate nothing was found. */

262. 345         current_ptr = TX_NULL;

263. 346     }

264. 347 }

265. 348

266. 349 /* Return the search pointer. */

267. 350 return(current_ptr);

```



## 2.4、byte内存释放\_tx\_byte\_release

---

释放内存过程比较简单，与申请内存有很多重复的地方：

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

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

```
1. 077 UINT    _tx_byte_release(VOID *memory_ptr)

2. 078 {

3. 079

4. 080 TX_INTERRUPT_SAVE_AREA

5. 081

6. 082 UINT                status;

7. 083 TX_BYTE_POOL        *pool_ptr;

8. 084 TX_THREAD            *thread_ptr;

9. 085 UCHAR                *work_ptr;

10. 086 UCHAR                *temp_ptr;

11. 087 UCHAR                *next_block_ptr;

12. 088 TX_THREAD            *susp_thread_ptr;

13. 089 UINT                suspended_count;

14. 090 TX_THREAD            *next_thread;

15. 091 TX_THREAD            *previous_thread;

16. 092 ULONG                memory_size;

17. 093 ALIGN_TYPE            *free_ptr;

18. 094 TX_BYTE_POOL        **byte_pool_ptr;

19. 095 UCHAR                **block_link_ptr;

20. 096 UCHAR                **suspend_info_ptr;

21. 097

22. 098

23. 099     /* Default to successful status.  */

24. 100     status = TX_SUCCESS;

25. 101

26. 102     /* Set the pool pointer to NULL.  */

27. 103     pool_ptr = TX_NULL;

28. 104

29. 105     /* Lockout interrupts.  */

30. 106     TX_DISABLE

31. 107
```



```

32. 108      /* Determine if the memory pointer is valid.  */
33. 109      work_ptr = TX_VOID_TO_UCHAR_POINTER_CONVERT(memory_ptr);
34. 110      if (work_ptr != TX_NULL) // 检查释放的内存是否为空指针(if分支主要获取
    memory_ptr所在的pool)
35. 111      {
36. 112
37. 113          /* Back off the memory pointer to pickup its header.  */
38. 114          work_ptr = TX_UCHAR_POINTER_SUB(work_ptr, ((sizeof(UCHAR *) +
    (sizeof(ALIGN_TYPE)))); // 获取内存块的起始地址
39. 115
40. 116          /* There is a pointer, pickup the pool pointer address.  */
41. 117          temp_ptr = TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR *))); // 内存块
    起始地址加上下一个内存地址大小就是pool指针/free标志所在内存
42. 118          free_ptr = TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(temp_ptr);
43. 119          if ((*free_ptr) != TX_BYTE_BLOCK_FREE) // 检查内存是否是空闲内存，如果重
    复释放内存，返回TX_PTR_ERROR
44. 120          {
45. 121
46. 122          /* Pickup the pool pointer.  */
47. 123          temp_ptr = TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR *))); // 获
    取pool指针所在内存地址，内存块结构为"|下一个内存块地址|pool指针/free标
    志|memory_ptr|"
48. 124          byte_pool_ptr = TX_UCHAR_TO_INDIRECT_BYTE_POOL_POINTER(temp_ptr);
    // 指针转换
49. 125          pool_ptr = *byte_pool_ptr; // 获取pool指针
50. 126
51. 127          /* See if we have a valid pool pointer.  */
52. 128          if (pool_ptr == TX_NULL) // pool为空，那么memory_ptr不是pool里面的内
    存块，使用了错误的释放函数
53. 129          {
54. 130
55. 131          /* Return pointer error.  */
56. 132          status = TX_PTR_ERROR; // 返回错误
57. 133          }
58. 134          else // pool指针不为空，需要再次检查是否是真正的pool
59. 135          {

```

```

60. 136
61. 137             /* See if we have a valid pool.  */
62. 138             if (pool_ptr -> tx_byte_pool_id != TX_BYTE_POOL_ID) // pool id对
            不上，pool指针不是真正指向有效的pool，返回错误
63. 139             {
64. 140
65. 141             /* Return pointer error.  */
66. 142             status = TX_PTR_ERROR;
67. 143
68. 144             /* Reset the pool pointer is NULL.  */
69. 145             pool_ptr = TX_NULL;
70. 146         }
71. 147     }
72. 148 }
73. 149 else
74. 150 {
75. 151
76. 152     /* Return pointer error.  */
77. 153     status = TX_PTR_ERROR;
78. 154 }
79. 155 }
80. 156 else
81. 157 {
82. 158
83. 159     /* Return pointer error.  */
84. 160     status = TX_PTR_ERROR;
85. 161 }
86. 162
87. 163 /* Determine if the pointer is valid.  */
88. 164 if (pool_ptr == TX_NULL) // 没有找到memory_ptr所在的有效pool，memory_ptr不是
    从pool申请的，返回错误
89. 165 {
90. 166

```

```

91. 167      /* Restore interrupts. */
92. 168      TX_RESTORE
93. 169  }
94. 170  else // 释放memory_ptr
95. 171  {
96. 172
97. 173      /* At this point, we know that the pointer is valid. */
98. 174
99. 175      /* Pickup thread pointer. */
100. 176      TX_THREAD_GET_CURRENT(thread_ptr)
101. 177
102. 178      /* Indicate that this thread is the current owner. */
103. 179      pool_ptr -> tx_byte_pool_owner = thread_ptr; // 标记当前线程正在操作
pool(与申请内存作用一样)
104. 180
105. 181 #ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO
106. 182
107. 183      /* Increment the total release counter. */
108. 184      _tx_byte_pool_performance_release_count++;
109. 185
110. 186      /* Increment the number of releases on this pool. */
111. 187      pool_ptr -> tx_byte_pool_performance_release_count++;
112. 188 #endif
113. 189
114. 190      /* If trace is enabled, insert this event into the trace buffer. */
115. 191      TX_TRACE_IN_LINE_INSERT(TX_TRACE_BYTE_RELEASE, pool_ptr,
TX_POINTER_TO_ULONG_CONVERT(memory_ptr), pool_ptr -> tx_byte_pool_suspended_count,
pool_ptr -> tx_byte_pool_available, TX_TRACE_BYTE_POOL_EVENTS)
116. 192
117. 193      /* Log this kernel call. */
118. 194      TX_EL_BYTE_RELEASE_INSERT
119. 195
120. 196      /* Release the memory. */

```

```

121. 197         temp_ptr =    TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR *)));
122. 198         free_ptr =    TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(temp_ptr);
123. 199         *free_ptr =    TX_BYTE_BLOCK_FREE; // 将内存块标记为空闲
124. 200
125. 201         /* Update the number of available bytes in the pool. */
126. 202         block_link_ptr = TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr);
127. 203         next_block_ptr = *block_link_ptr;
128. 204         pool_ptr -> tx_byte_pool_available =
129. 205             pool_ptr -> tx_byte_pool_available +
130. 206             TX_UCHAR_POINTER_DIF(next_block_ptr, work_ptr); //
131. 207             "TX_UCHAR_POINTER_DIF(next_block_ptr, work_ptr)"当前内存块与下一个内存块之间的距离
132. 208             (包括下一个内存块地址指针等字段)作为当前空闲内存块的大小，当前释放的空闲内存块可能与
133. 209             相邻的空闲内存块合并，下一个内存地址等字段可能变成有效内存
134. 210
135. 211         /* Determine if the free block is prior to current search pointer. */
136. 212         if (work_ptr < (pool_ptr -> tx_byte_pool_search)) // 释放的空闲内存块地
137. 213             址小于下一次查找空闲内存块的起始地址，更新tx_byte_pool_search指向当前释放的空闲内存
138. 214             块；ThreadX尽量从低地址开始分配内存，这样位置越后的空闲内存块被拆分的机会就更少，也
139. 215             就是内存块就越大，有大内存申请的时候才可能获取到大内存；tx_byte_pool_search也不指向
140. 216             内存块表头，这样影响查找效率，所以内核是尽可能从前面的空闲内存块开始查找
141. 217         {
142. 218
143. 219             /* Yes, update the search pointer to the released block. */
144. 220             pool_ptr -> tx_byte_pool_search = work_ptr; // 更新
145. 221             tx_byte_pool_search
146. 222         }
147. 223
148. 224         /* Determine if there are threads suspended on this byte pool. */
149. 225         if (pool_ptr -> tx_byte_pool_suspended_count != TX_NO_SUSPENSIONS) // 有
150. 226             线程在等待内存
151. 227         {
152. 228
153. 229             /* Now examine the suspension list to find threads waiting for
154. 230             memory. Maybe it is now available! */
155. 231             while (pool_ptr -> tx_byte_pool_suspended_count !=
156. 232                 TX_NO_SUSPENSIONS) // 循环给表头线程分配内存(表头线程分配到内存后，下一个等待内存的
157. 233                 线程将成为新的表头，循环过程就是一次给等待线程分配内存，如果分配不到就退出循环，不给
158. 234                 后续等待线程分配内存)

```

```

146. 222      {
147. 223
148. 224          /* Pickup the first suspended thread pointer. */
149. 225      susp_thread_ptr = pool_ptr -> tx_byte_pool_suspension_list; //
      等待内存的线程链表
150. 226
151. 227          /* Pickup the size of the memory the thread is requesting. */
152. 228      memory_size = susp_thread_ptr -> tx_thread_suspend_info; // 等
      待内存的大小(也就是malloc的内存大小)
153. 229
154. 230          /* Restore interrupts. */
155. 231      TX_RESTORE // 恢复中断, 已经获取到了一个等待内存线程的数据, 先试
      图给这个线程分配内存, 别的线程还可以继续申请内存(tx_byte_pool_suspension_list是所有
      申请内存的线程共同访问的, 关闭中断将导致其他线程不能申请内存)
156. 232
157. 233          /* See if the request can be satisfied. */
158. 234      work_ptr = _tx_byte_pool_search(pool_ptr, memory_size); // 与申
      请内存时一样, 查找memory_size的空闲内存块
159. 235
160. 236          /* Optional processing extension. */
161. 237      TX_BYTE_RELEASE_EXTENSION
162. 238
163. 239          /* Disable interrupts. */
164. 240      TX_DISABLE
165. 241
166. 242          /* Indicate that this thread is the current owner. */
167. 243      pool_ptr -> tx_byte_pool_owner = thread_ptr;
168. 244
169. 245          /* If there is not enough memory, break this loop! */
170. 246      if (work_ptr == TX_NULL) // 没有找到可用内存块(这里不用判断是否
      有其他线程也操作了内存, 如果别的线程释放了内存, 那么也会试图给阻塞线程分配内存)
171. 247      {
172. 248
173. 249          /* Break out of the loop. */

```

```

174. 250                break; // 退出循环，不继续分配内存(从这里看，ThreadX内核分配
                        内存是优先给等待链表前面的线程分配内存，如果前面的线程都分配不到内存，那么就不给后面
                        等待内存线程分配内存，即使内存够大)

175. 251                }

176. 252

177. 253                /* Check to make sure the thread is still suspended. */

178. 254                if (susp_thread_ptr == pool_ptr ->
tx_byte_pool_suspension_list) // 再次检查susp_thread_ptr是否在阻塞链表里面(新的等待
内存的线程会挂到等待链表末尾，不会改变表头，只有等待内存的线程超时，被从等待链表删除
时，表头才可能变成其他线程)

179. 255                {

180. 256

181. 257                /* Also, makes sure the memory size is the same. */

182. 258                if (susp_thread_ptr -> tx_thread_suspend_info ==
memory_size) // 再次检查内存大小是否是刚才查找内存块的大小(存在等待内存的线程超时后
再次加入等待链表的情况，也就是查找过程所有等待内存的线程都超时或者其他情况导致等待内
存的线程链表都清空了，当前线程还没检测到，然后之前等待内存链表表头的线程又再次申请内
存并且阻塞了，那么该线程又回到了阻塞链表的表头)

183. 259                {

184. 260

185. 261                /* Remove the suspended thread from the list. */

186. 262

187. 263                /* Decrement the number of threads suspended. */

188. 264                pool_ptr -> tx_byte_pool_suspended_count--; // 等待内存
                        的线程数量减1

189. 265

190. 266                /* Pickup the suspended count. */

191. 267                suspended_count = pool_ptr ->
tx_byte_pool_suspended_count;

192. 268

193. 269                /* See if this is the only suspended thread on the list.
*/

194. 270                if (suspended_count == TX_NO_SUSPENSIONS) // 如果没有其
                        他线程等待内存，清空等待链表

195. 271                {

196. 272

197. 273                /* Yes, the only suspended thread. */

198. 274

```

```

199. 275                                /* Update the head pointer. */
200. 276                                pool_ptr -> tx_byte_pool_suspension_list = TX_NULL;
    // 清空等待链表
201. 277                                }
202. 278                                else // 否则从等待链表删除当前分配内存的线程(表头线程)
203. 279                                {
204. 280
205. 281                                /* At least one more thread is on the same
    expiration list. */
206. 282
207. 283                                /* Update the list head pointer. */
208. 284                                next_thread =
    susp_thread_ptr -> tx_thread_suspended_next;
209. 285                                pool_ptr -> tx_byte_pool_suspension_list =
    next_thread;
210. 286
211. 287                                /* Update the links of the adjacent threads. */
212. 288                                previous_thread =
    susp_thread_ptr -> tx_thread_suspended_previous;
213. 289                                next_thread -> tx_thread_suspended_previous =
    previous_thread;
214. 290                                previous_thread -> tx_thread_suspended_next =
    next_thread;
215. 291                                }
216. 292
217. 293                                /* Prepare for resumption of the thread. */
218. 294
219. 295                                /* Clear cleanup routine to avoid timeout. */
220. 296                                susp_thread_ptr -> tx_thread_suspend_cleanup = TX_NULL;
    // 清空tx_thread_suspend_cleanup, 线程已经获取到内存了(这里还没去激活超时定时器, 清
    空tx_thread_suspend_cleanup, 即使定时器超时也不会有回调函数调用)
221. 297
222. 298                                /* Return this block pointer to the suspended thread
    waiting for
223. 299                                a block. */
224. 300                                suspend_info_ptr =
    TX_VOID_TO_INDIRECT_UCHAR_POINTER_CONVERT(susp_thread_ptr ->
    tx_thread_additional_suspend_info); // 申请内存时的memory_ptr指针

```

```

225. 301          *suspend_info_ptr = work_ptr; // 设置memory_ptr, 将
           work_ptr内存分配给线程

226. 302

227. 303          /* Clear the memory pointer to indicate that it was
           given to the suspended thread. */

228. 304          work_ptr = TX_NULL; // 清空work_ptr(指示该内存已经被分
           配了, 后面会用到work_ptr, 如果等待内存链表表头变了或者内存大小不一致, 那么需要将该内
           存块重新设置为空闲内存)

229. 305

230. 306          /* Put return status into the thread control block. */

231. 307          susp_thread_ptr -> tx_thread_suspend_status =
           TX_SUCCESS; // 线程返回状态(阻塞线程被唤醒后, 使用这个作为获取内存的结果返回给上一级
           函数)

232. 308

233. 309 #ifdef TX_NOT_INTERRUPTABLE

234. 310

235. 311          /* Resume the thread! */

236. 312          _tx_thread_system_ni_resume(susp_thread_ptr);

237. 313

238. 314          /* Restore interrupts. */

239. 315          TX_RESTORE

240. 316 #else

241. 317          /* Temporarily disable preemption. */

242. 318          _tx_thread_preempt_disable++;

243. 319

244. 320          /* Restore interrupts. */

245. 321          TX_RESTORE

246. 322

247. 323          /* Resume thread. */

248. 324          _tx_thread_system_resume(susp_thread_ptr); // 唤醒已经分
           配到内存的线程

249. 325 #endif

250. 326

251. 327          /* Lockout interrupts. */

252. 328          TX_DISABLE

```



```

253. 329                }

254. 330                }

255. 331

256. 332                /* Determine if the memory was given to the suspended thread.
    */

257. 333                if (work_ptr != TX_NULL) // 等待内存的表头线程被改变了(内存没有
    分配成功, 需要将内存重新变成空闲状态)

258. 334                {

259. 335

260. 336                /* No, it wasn't given to the suspended thread.  */

261. 337

262. 338                /* Put the memory back on the available list since this
    thread is no longer

263. 339                suspended.  */

264. 340                work_ptr = TX_UCHAR_POINTER_SUB(work_ptr, (((sizeof(UCHAR
    *)) + (sizeof(ALIGN_TYPE)))));

265. 341                temp_ptr = TX_UCHAR_POINTER_ADD(work_ptr, (sizeof(UCHAR
    *)));

266. 342                free_ptr =
    TX_UCHAR_TO_ALIGN_TYPE_POINTER_CONVERT(temp_ptr);

267. 343                *free_ptr = TX_BYTE_BLOCK_FREE;

268. 344

269. 345                /* Update the number of available bytes in the pool.  */

270. 346                block_link_ptr =
    TX_UCHAR_TO_INDIRECT_UCHAR_POINTER_CONVERT(work_ptr);

271. 347                next_block_ptr = *block_link_ptr;

272. 348                pool_ptr -> tx_byte_pool_available =

273. 349                pool_ptr -> tx_byte_pool_available +
    TX_UCHAR_POINTER_DIF(next_block_ptr, work_ptr);

274. 350

275. 351                /* Determine if the current pointer is before the search
    pointer.  */

276. 352                if (work_ptr < (pool_ptr -> tx_byte_pool_search))

277. 353                {

278. 354

279. 355                /* Yes, update the search pointer.  */

```

```
280. 356                pool_ptr -> tx_byte_pool_search = work_ptr; // work_ptr
    重新释放，更新tx_byte_pool_search

281. 357                }

282. 358            }

283. 359        }

284. 360

285. 361        /* Restore interrupts. */

286. 362        TX_RESTORE

287. 363

288. 364        /* Check for preemption. */

289. 365        _tx_thread_system_preempt_check();

290. 366    }

291. 367    else // 没有线程等待内存，释放内存块标记为空闲即可，返回上一级函数

292. 368    {

293. 369

294. 370        /* No, threads suspended, restore interrupts. */

295. 371        TX_RESTORE

296. 372    }

297. 373 }

298. 374

299. 375 /* Return completion status. */

300. 376 return(status);

301. 377 }
```

