(163条消息) ThreadX内核源码分析 - 计数信号量_arm7star 的博客-CSDN博客

blog.csdn.net/arm7star/article/details/123442459

1、计数信号量介绍

计数信号量的信号量值不为0,表示信号量可获取,每次获取信号量,信号量计数器的值减 1,为0是,信号量不看获取,释放信号量是每次加1;

计数信号量实现与互斥锁类似,一定程度上可以把互斥锁看出计数为1的信号量,只不过互 斥锁有动态优先级调整,信号量没有,互斥锁用于临界资源保护,信号量用于生产消费者 这类场景(多个消费者、多个生产者)。

2、信号量获取 tx semaphore get

获取信号量的代码比较简单, 主要是对计数信号量的计数进行判断, 如果信号量计数器不 为o, 就减1然后返回(获取到了信号量), 如果信号量计数器为o(获取不到信号量), 那么设 置自己的阻塞状态,设置超时/终止等情况下的清理函数,设置阻塞的信号量,将自己加入 到信号量的阻塞队列,挂起自己,内核的挂起函数在挂起当前线程是会选择下一个执行线 程并进行线程切换(切换过程当然也会保存当前线程的上下文)。

tx semaphore get实现代码如下:

```
    076 UINT _tx_semaphore_get(TX_SEMAPHORE *semaphore_ptr, ULONG wait_option)

 2. 077 {
 3. 078
4. 079 TX INTERRUPT SAVE AREA
 5. 080
 6. 081 TX THREAD
                        *thread ptr;
7. 082 TX THREAD
                        *next thread;
 8. 083 TX THREAD
                        *previous thread;
9. 084 UINT
                        status;
10.085
11. 086
12. 087
            /* Default the status to TX SUCCESS. */
13. 088
            status = TX_SUCCESS;
14. 089
15. 090
           /* Disable interrupts to get an instance from the semaphore. */
16. 091
            TX DISABLE
17. 092
18. 093 #ifdef TX SEMAPHORE ENABLE PERFORMANCE INFO
19. 094
20. 095
            /* Increment the total semaphore get counter. */
21. 096
            _tx_semaphore_performance_get_count++;
22. 097
23. 098
            /* Increment the number of attempts to get this semaphore. */
24. 099
            semaphore_ptr -> tx_semaphore_performance_get_count++;
25. 100 #endif
26. 101
27. 102
            /* If trace is enabled, insert this event into the trace buffer. */
28. 103
            TX_TRACE_IN_LINE_INSERT(TX_TRACE_SEMAPHORE_GET, semaphore_ptr, wait_option,
    semaphore_ptr -> tx_semaphore_count, TX_POINTER_TO_ULONG_CONVERT(&thread_ptr),
    TX_TRACE_SEMAPHORE_EVENTS)
29. 104
30. 105
          /* Log this kernel call. */
```

```
31. 106
          TX_EL_SEMAPHORE_GET_INSERT
32. 107
33. 108
          /* Determine if there is an instance of the semaphore. */
34. 109
          if (semaphore_ptr -> tx_semaphore_count != ((ULONG) 0)) // 信号量计数器不为
   0,可以获取到信号量
35. 110
          {
36. 111
37. 112
              /* Decrement the semaphore count. */
38. 113
              semaphore ptr -> tx semaphore count--; // 简单对计数信号量减1即可, 然后
   返回成功(与互斥锁不同, 计数信号量没有记录owner)
39. 114
40. 115
              /* Restore interrupts. */
41. 116
             TX RESTORE
42. 117
          }
43. 118
44. 119
          /* Determine if the request specifies suspension. */
          else if (wait_option != TX_NO_WAIT) // 信号量为0, 获取不到信号量, 并且
   wait option不是TX NO WAIT,也就是获取信号量是阻塞的
46. 121
47. 122
48. 123
              /* Determine if the preempt disable flag is non-zero. */
              if (_tx_thread_preempt_disable != ((UINT) 0)) // 如果禁止了抢占,那么不
   能阻塞获取不到信号量的线程, 否则所有其他就绪线程都得不到调度
50. 125
              {
51. 126
                 /* Restore interrupts. */
52. 127
53. 128
                 TX RESTORE
54. 129
55. 130
                 /* Suspension is not allowed if the preempt disable flag is non-zero
   at this point - return error completion. */
56. 131
                 status = TX_NO_INSTANCE; // 获取不到信号量,禁止抢占,返回
   TX_NO_INSTANCE即可
57. 132
              }
              else // 没有禁止抢占, 获取不到信号量的线程可以进入阻塞状态
58. 133
59. 134
              {
```

```
60. 135
61. 136
                  /* Prepare for suspension of this thread. */
62. 137
63. 138 #ifdef TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO
64. 139
                  /* Increment the total semaphore suspensions counter. */
65, 140
66. 141
                  tx semaphore performance suspension count++;
67. 142
68. 143
                  /* Increment the number of suspensions on this semaphore. */
69. 144
                  semaphore ptr -> tx semaphore performance suspension count++;
70. 145 #endif
71, 146
72, 147
                 /* Pickup thread pointer. */
73. 148
                  TX THREAD GET CURRENT(thread ptr) // 获取当前线程
   _tx_thread_current_ptr
74. 149
75. 150
                  /* Setup cleanup routine pointer. */
                  thread_ptr -> tx_thread_suspend_cleanup = &(_tx_semaphore_cleanup);
76. 151
   // 设置超时清理函数 tx semaphore cleanup(与前面文章介绍的获取内存阻塞一样,超时需要
   通过 tx semaphore cleanup唤醒线程并且从等待信号量队列删除线程)
77. 152
78. 153
                 /* Setup cleanup information, i.e. this semaphore control
79. 154
                     block. */
80. 155
                  thread_ptr -> tx_thread_suspend_control_block = (VOID *)
   semaphore_ptr; // 阻塞在信号量semaphore_ptr上(线程挂在semaphore_ptr等待链表上,
   _tx_semaphore_cleanup及释放信号量的线程需要通过semaphore_ptr找到等待信号量的线程)
81. 156
82. 157 #ifndef TX NOT INTERRUPTABLE
83. 158
84. 159
                  /* Increment the suspension sequence number, which is used to
   identify
85. 160
                    this suspension event. */
86. 161
                  thread_ptr -> tx_thread_suspension_sequence++;
87. 162 #endif
88. 163
```

```
89. 164
                    /* Setup suspension list. */
90. 165
                    if (semaphore_ptr -> tx_semaphore_suspended_count ==
    TX_NO_SUSPENSIONS) // 只有当前线程等待信号量,加入tx_semaphore_suspension_list信号量
    等待链表
91. 166
                    {
92. 167
93. 168
                        /* No other threads are suspended. Setup the head pointer and
94. 169
                           just setup this threads pointers to itself. */
95. 170
                        semaphore ptr -> tx semaphore suspension list =
    thread ptr;
96. 171
                        thread_ptr -> tx_thread_suspended_next =
    thread_ptr;
97. 172
                        thread_ptr -> tx_thread_suspended_previous =
    thread ptr;
98. 173
                    }
99. 174
                    else // 有多个线程等待信号量,加入tx semaphore suspension list信号量
    等待链表,添加到末尾
100. 175
101. 176
                        /* This list is not NULL, add current thread to the end. */
102. 177
103. 178
                        next_thread =
                                                                       semaphore ptr ->
    tx semaphore suspension list;
104. 179
                        thread ptr -> tx thread suspended next =
                                                                       next thread;
105. 180
                        previous_thread =
                                                                       next thread ->
    tx thread suspended previous;
106. 181
                        thread ptr -> tx thread suspended previous =
                                                                      previous thread;
107. 182
                        previous_thread -> tx_thread_suspended_next =
                                                                       thread_ptr;
108. 183
                        next thread -> tx thread suspended previous =
                                                                       thread ptr;
109. 184
                    }
110. 185
111. 186
                    /* Increment the number of suspensions. */
112. 187
                    semaphore_ptr -> tx_semaphore_suspended_count++; // 等待信号量的线程
    个数加1
113. 188
114. 189
                    /* Set the state to suspended. */
```

```
thread_ptr -> tx_thread_state = TX_SEMAPHORE_SUSP; // 设置当前线
115. 190
    程的状态,等待信号量超时需要判断状态,其他一些唤醒阻塞操作也需要判断状态(一些状态的
    线程不能被唤醒或者挂起,或者说线程处于不可中断的阻塞状态)
116. 191
117. 192 #ifdef TX_NOT_INTERRUPTABLE
118. 193
119. 194
                /* Call actual non-interruptable thread suspension routine. */
120. 195
                 _tx_thread_system_ni_suspend(thread_ptr, wait_option);
121. 196
122. 197
                 /* Restore interrupts. */
123. 198
                 TX RESTORE
124. 199 #else
125. 200
126. 201
                /* Set the suspending flag. */
127. 202
                 thread ptr -> tx thread suspending = TX TRUE; // 设置
    tx thread suspending,线程还没从就绪链表删除
128. 203
129. 204
                 /* Setup the timeout period. */
130. 205
                 thread ptr -> tx thread timer.tx timer internal remaining ticks =
    wait option; // 等待超时时间,如果wait option不是无限等待,那么
    tx thread system suspend挂起线程时就会启动一个定时器,超时调用
    tx thread suspend cleanup唤醒线程,然后调用 tx semaphore cleanup处理信号量超时事件
131. 206
132. 207
                 /* Temporarily disable preemption. */
                 _tx_thread_preempt_disable++; // 禁止抢占(_tx_thread_system_suspend
133. 208
    会对_tx_thread_preempt_disable减1,调用_tx_thread_system_suspend前必须对
    _tx_thread_preempt_disable加1)
134. 209
135. 210
                 /* Restore interrupts. */
136. 211
                 TX_RESTORE // 允许中断(之前对抢占计数器加1了,
    tx thread system suspend执行期间除了能处理中断服务程序外,其他线程是不允许被调度执
    行的,禁止抢占加上允许中断的目的只是避免阻塞中断并让 tx thread system suspend执行完)
137. 212
138. 213
                 /* Call actual thread suspension routine. */
139. 214
                 _tx_thread_system_suspend(thread_ptr); // 挂起当前线程
140. 215 #endif
```

141. 216

```
143. 218 status = thread_ptr -> tx_thread_suspend_status; // 等待超时,清理函数会设置tx_thread_suspend_status为超时; 别的线程释放信号量会释放给当前线程,tx_thread_suspend_status会设置为成功,当前线程被唤醒后,根据tx_thread_suspend_status即可知道自己是获取到了信号量还是超时了
```

/* Return the completion status. */

```
144. 219
                 }
145. 220
             }
146. 221
             else
147. 222
             {
148, 223
149. 224
                /* Restore interrupts. */
150. 225
                TX RESTORE
151. 226
152. 227
                /* Immediate return, return error completion. */
153. 228
                 status = TX NO INSTANCE;
154. 229
            }
155. 230
156. 231
           /* Return completion status. */
157. 232
            return(status);
158. 233 }
```

142. 217

至于等待信号量超时,在此略过,处理比较简单,与申请内存超时实现基本一样,可以参考:

ThreadX内核源码分析 - 动态内存管理_arm7star的博客-CSDN博客_threadx 内存管理

3、信号量释放_tx_semaphore_put

释放信号量操作也比较简单,主要是对计数信号量的计数器加1,然后检查是否有线程等待信号量,如果没有,对信号量计数器加1,直接返回即可,如果有等待信号量的线程,将信号量给等待链表第一个等待信号量的线程并唤醒该线程即可(不对信号量加1,直接把该信号量给等待线程,等待信号量的线程阻塞唤醒后不会再对信号量减1,可能有的内核会恢复信号量的值,然后唤醒所有等待信号量的线程,让等待信号量的线程去重新抢信号量,ThreadX内核没有这么做!!!)。

_tx_semaphore_put实现代码如下:

```
1. 075 UINT _tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr)
 2. 076 {
 3. 077
 4. 078 TX INTERRUPT SAVE AREA
 5. 079
 6. 080 #ifndef TX DISABLE NOTIFY CALLBACKS
 7. 081 VOID
                        (*semaphore put notify)(struct TX SEMAPHORE STRUCT
    *notify_semaphore_ptr);
 8. 082 #endif
 9. 083
10. 084 TX_THREAD
                        *thread 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 an instance back to the semaphore. */
17. 091
           TX DISABLE // 美闭中断
18. 092
19. 093 #ifdef TX SEMAPHORE ENABLE PERFORMANCE INFO
20. 094
21. 095
           /* Increment the total semaphore put counter. */
22. 096
            tx semaphore performance put count++;
23. 097
24. 098
            /* Increment the number of puts on this semaphore. */
25. 099
            semaphore_ptr -> tx_semaphore_performance_put_count++;
26. 100 #endif
27. 101
28. 102
           /* If trace is enabled, insert this event into the trace buffer. */
            TX_TRACE_IN_LINE_INSERT(TX_TRACE_SEMAPHORE_PUT, semaphore_ptr, semaphore_ptr
29. 103
    -> tx_semaphore_count, semaphore_ptr -> tx_semaphore_suspended_count,
    TX_POINTER_TO_ULONG_CONVERT(&thread_ptr), TX_TRACE_SEMAPHORE_EVENTS)
```

30. 104

```
31. 105
          /* Log this kernel call. */
32. 106
          TX_EL_SEMAPHORE_PUT_INSERT
33. 107
34. 108
          /* Pickup the number of suspended threads. */
35. 109
           suspended_count = semaphore_ptr -> tx_semaphore_suspended_count; // 等待信
   号量的线程数量
36. 110
37. 111
          /* Determine if there are any threads suspended on the semaphore. */
38. 112
          if (suspended count == TX NO SUSPENSIONS) // 没有线程等待信号量
39. 113
40. 114
41. 115
               /* Increment the semaphore count. */
42. 116
               semaphore ptr -> tx semaphore count++; // 信号量计数器加1, 恢复信号量计
   数器即可
43. 117
44. 118 #ifndef TX DISABLE NOTIFY CALLBACKS
45. 119
              /* Pickup the application notify function. */
46. 120
47. 121
               semaphore_put_notify = semaphore_ptr -> tx_semaphore_put_notify;
48. 122 #endif
49. 123
50. 124
            /* Restore interrupts. */
51. 125
              TX RESTORE
52. 126
53. 127 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
54. 128
55. 129
              /* Determine if notification is required. */
56. 130
              if (semaphore_put_notify != TX_NULL)
57. 131
               {
58. 132
59. 133
                   /* Yes, call the appropriate notify callback function. */
60. 134
                   (semaphore_put_notify)(semaphore_ptr);
61. 135
               }
```

```
63. 137
          }
64. 138
         else // 有线程等待信号量(此次信号量计数器还没被修改,信号量还没释放)
65. 139
          {
66. 140
             /* A thread is suspended on this semaphore. */
67. 141
68. 142
69. 143
             /* Pickup the pointer to the first suspended thread. */
             thread ptr = semaphore ptr -> tx semaphore suspension list; // 获取第一
70. 144
   个等待信号量的线程(与互斥锁场景不同,继承优先级的互斥锁是让高优先级线程先获取到互斥
   锁,避免高优先级线程等待低优先级线程的情况)
71. 145
72. 146
             /* Remove the suspended thread from the list. */
73. 147
74. 148
             /* See if this is the only suspended thread on the list. */
75. 149
             suspended count--; // 等待信号量的线程数量减1
76. 150
             if (suspended_count == TX_NO_SUSPENSIONS) // 如果没有其他线程等待信号
   量,那么情况信号量的等待链表即可
77. 151
             {
78. 152
79. 153
                 /* Yes, the only suspended thread. */
80. 154
81. 155
                 /* Update the head pointer. */
82. 156
                 semaphore_ptr -> tx_semaphore_suspension_list = TX_NULL;
83. 157
             }
84. 158
             else // 有其他线程等待信号量,第一个等待信号量的线程从等待链表删除即可
   (第一个等待信号量的线程thread ptr即将获取到信号量)
85. 159
             {
86. 160
87. 161
                 /* At least one more thread is on the same expiration list. */
88. 162
89. 163
                 /* Update the list head pointer. */
90. 164
                 next_thread =
                                                            thread_ptr ->
   tx_thread_suspended_next;
```

62. 136 #endif

```
91. 165
                   semaphore_ptr -> tx_semaphore_suspension_list = next_thread;
92. 166
93. 167
                   /* Update the links of the adjacent threads. */
94. 168
                   previous_thread =
                                                               thread ptr ->
    tx_thread_suspended_previous;
95. 169
                   next thread -> tx thread suspended previous =
                                                               previous thread;
96, 170
                   previous thread -> tx thread suspended next =
                                                               next thread;
97. 171
               }
98, 172
99. 173
               /* Decrement the suspension count. */
               semaphore_ptr -> tx_semaphore_suspended_count = suspended_count; // 更
100. 174
    新信号量等待线程个数(suspended count在前面减1了,减thread ptr)
101. 175
102. 176
               /* Prepare for resumption of the first thread. */
103. 177
104. 178
               /* Clear cleanup routine to avoid timeout. */
               thread_ptr -> tx_thread_suspend_cleanup = TX_NULL; // 清空thread_ptr的
105. 179
    tx thread suspend cleanup(thread ptr获取到了信号量,不再需要清理函数)
106. 180
107. 181 #ifndef TX DISABLE NOTIFY CALLBACKS
108. 182
109. 183
               /* Pickup the application notify function. */
110. 184
               semaphore_put_notify = semaphore_ptr -> tx_semaphore_put_notify;
111. 185 #endif
112. 186
113. 187
               /* Put return status into the thread control block. */
               thread_ptr -> tx_thread_suspend_status = TX_SUCCESS; // thread_ptr获取
114. 188
    到信号量,tx_thread_suspend_status设置为TX_SUCCESS,thread_ptr唤醒后会用
    tx_thread_suspend_status作为返回值,用于判断是否获取到信号量(如果超时,超时函数会设
    置tx_thread_suspend_status为超时状态,阻塞的线程唤醒后并不知道自己唤醒的原因,因此需
    要通过tx_thread_suspend_status来判断自己是怎么被唤醒的)
115. 189
116. 190 #ifdef TX_NOT_INTERRUPTABLE
117. 191
118. 192
             /* Resume the thread! */
```

```
119. 193
                _tx_thread_system_ni_resume(thread_ptr);
120. 194
121. 195
               /* Restore interrupts. */
122. 196
               TX_RESTORE
123. 197 #else
124. 198
125. 199
               /* Temporarily disable preemption. */
126. 200
                _tx_thread_preempt_disable++; // 禁止抢占(_tx_thread_system_resume会对抢
     占计数器减1,这里必须加1)
127. 201
128. 202
            /* Restore interrupts. */
129. 203
               TX RESTORE // 允许中断
130. 204
131. 205
              /* Resume thread. */
                _tx_thread_system_resume(thread_ptr); // 唤醒thread_ptr(如果有抢占的话,
132. 206
    会发生线程切换)
133. 207 #endif
134. 208
135. 209 #ifndef TX_DISABLE_NOTIFY_CALLBACKS
136. 210
137. 211
               /* Determine if notification is required. */
138. 212
               if (semaphore_put_notify != TX_NULL)
139. 213
               {
140. 214
141. 215
                    /* Yes, call the appropriate notify callback function. */
142. 216
                    (semaphore_put_notify)(semaphore_ptr);
143. 217
                }
144. 218 #endif
145. 219
            }
146. 220
147. 221
           /* Return successful completion. */
148. 222
           return(TX_SUCCESS);
149. 223 }
```

