(163条消息) ThreadX内核源码分析(SMP) - 线程多核映射_arm7star的博客-CSDN博客_threadx

1、线程remap介绍(_tx_thread_smp_remap_solution_find)

内核在挂起正在核上执行的线程时，会空出一个核，那么就可能在该核上执行新的线程，另外，因为是多核，所以线程存在绑核的情况，也就是新的线程可能绑定到其他核了，并不能在空出的核上执行，而其他核又有其他线程在执行，那么就需要考虑是否可以移动其他核的线程到空闲核上，然后让新的线程在让出的核上执行，这就涉及到一个移动线程的操作(remap)；

内核唤醒一个新线程时，如果有空闲的核并且该线程可以在该核上执行，那么就可以让新线程在该核上执行，如果有空闲核但是不能在该核执行，那么需要考虑是否可以移动其他核上的线程到空闲核上，让出核给新的线程执行，如果没有空闲核并且新线程可以抢占正在执行的线程，那么也要淘汰一个线程，空出一个核来执行，新线程并不一定能在淘汰线程所在核上执行，因此也需要通过移动线程找到一个合适的方案来执行新的线程；

ThreadX内核通过_tx_thread_smp_remap_solution_find函数来查找一个能让线程在cpu上执行的方案。

2、_tx_thread_smp_remap_solution_find函数参数

函数原型：static INLINE_DECLARE UINT _tx_thread_smp_remap_solution_find(TX_THREAD *schedule_thread, ULONG available_cores, ULONG thread_possible_cores, ULONG test_possible_cores)

函数参数：

schedule_thread	需要调度的线程，也就是要给schedule_thread线程找一个可执行的核
available_cores	空闲的核
thread_possible_cores	schedule_thread线程可能的核(schedule_thread绑定的核，并且这些核上有其他线程在执行，移动这些线程可能空出核让schedule_thread线程执行)
test_possible_cores	正在执行的线程可以移动过去的核(不包括schedule_thread可能执行的核)

3、_tx_thread_smp_remap_solution_find实现(schedule_thread线程映射)

3.1、主要数据

UINT core_queue[TX_THREAD_SMP_MAX_CORES-1];

core_queue是一个数组，里面存的是核的id，以schedule_thread为例，如果schedule_thread线程上次执行的核可用(有别的线程在执行，可能可以空出来)，那么就把这个核保存到core_queue的最前面，然后从小到大将所有可能的执行的核加入到core_queue队列里面去，也就是按优先执行的核依次入队列；

UINT queue_first, queue_last;

queue_first、queue_last用于指向队列的头和尾，也就core_queue的头尾索引；

TX_THREAD *thread_remap_list[TX_THREAD_SMP_MAX_CORES];

thread_remap_list是一个线程指针数组，存储remap过程中的线程，thread_remap_list[i]也就是将该线程映射到核i上去执行。

3.2、schedule_thread线程可执行核入队列

schedule_thread线程可执行核入队列主要代码如下：

```
1186     /* Setup the core queue indices.  */
1187     queue_first =  ((UINT) 0);
1188     queue_last =   ((UINT) 0);
1189
1190     /* Build a list of possible cores for this thread to execute on, starting
1191        with the previously mapped core.  */
1192     core =  schedule_thread -> tx_thread_smp_core_mapped; // 线程上一次执行所在的核
1193     if ((thread_possible_cores & (((ULONG) 1) << core)) != ((ULONG) 0)) // 之前执行的核可用，之前执行的核放在core_queue前面，优先考虑之前执行的核
1194     {
1195
1196         /* Remember this potential mapping.  */
1197         thread_remap_list[core] =   schedule_thread; // 把schedule_thread线程映射到core上
1198         core_queue[queue_last] =    core; // core入队列
1199
1200         /* Move to next slot.  */
1201         queue_last++; // 队列的末尾加1后移
1202
1203         /* Clear this core.  */
1204         thread_possible_cores =  thread_possible_cores & ~(((ULONG) 1) << core); // core已经入队列了，清除thread_possible_cores对应的core
1205     }
```

获取线程上一次执行所在的核，让线程尽可能在上一次执行所在的核上执行：

```
1192     core =  schedule_thread -> tx_thread_smp_core_mapped; // 线程上一次执行所在的核
```

如果上一次执行所在的核可用(上面有线程在执行，可能可以空出来)，那么将该核入队列并从可用核上面删除，将线程映射到该核(只是该线程可以在该核上面执行，但是该核上面的线程并不一定能移动到其他核上去，后面代码再检查该核上原来执行的线程是否可以移动到其他核上面去)：

```
1193     if ((thread_possible_cores & (((ULONG) 1) << core)) != ((ULONG) 0)) // 之前执行的核可用，之前执行的核放在core_queue前面，优先考虑之前执行的核
1194     {
1195
1196         /* Remember this potential mapping.  */
1197         thread_remap_list[core] =   schedule_thread; // 把schedule_thread线程映射到core上
1198         core_queue[queue_last] =    core; // core入队列
1199
1200         /* Move to next slot.  */
1201         queue_last++; // 队列的末尾加1后移
1202
1203         /* Clear this core.  */
1204         thread_possible_cores =  thread_possible_cores & ~(((ULONG) 1) << core); // core已经入队列了，清除thread_possible_cores对应的core
1205     }
```

3.3、其他可执行的核从小到大入队列

线程可以执行的核依次按id从小到大入队列，也就是后面会从小到大检查是否存在可行方案让出核来给新的线程执行，优先让新的线程在小的核上面执行，主要代码如下:

```
    /* Loop to add additional possible cores.  */

    while (thread_possible_cores != ((ULONG) 0)) // schedule_thread可放的核依次加入core_queue，并将schedule_thread放到thread_remap_list里面

    {

        /* Determine the first possible core.  */

        test_cores =  thread_possible_cores;

        TX_LOWEST_SET_BIT_CALCULATE(test_cores, core) // TX_LOWEST_SET_BIT_CALCULATE计算test_cores从最低位开始的第一个非0的二进制位(最低的为1的二进制位，也就是id最小的核)，最小的可用核保存到core

        /* Clear this core.  */

        thread_possible_cores =  thread_possible_cores & ~(((ULONG) 1) << core); // 在可用核里面清除core

        /* Remember this potential mapping.  */

        thread_remap_list[core] =  schedule_thread; // schedule_thread线程映射到核core

        core_queue[queue_last] =   core; // 核core入队列

        /* Move to next slot.  */

        queue_last++; // 队列的末尾加1后移

    }
```

获取最小的可执行核，thread_possible_cores的每个二进制位代表一个核，0表示不可用，1表示可用，例如二进制的第3位为1，那么就表示核3可以用：

```
        /* Determine the first possible core.  */

        test_cores =  thread_possible_cores;

        TX_LOWEST_SET_BIT_CALCULATE(test_cores, core) // TX_LOWEST_SET_BIT_CALCULATE计算test_cores从最低位开始的第一个非0的二进制位(最低的为1的二进制位，也就是id最小的核)，最小的可用核保存到core
```

4、_tx_thread_smp_remap_solution_find实现(移动到空闲核)

因为新的线程不能直接在空闲核上面执行，那么就要移动其他正在执行的线程到空闲核上，空出一个核来给新的线程执行；上一节的schedule_thread线程映射过程中，schedule_thread线程映射到了所有可能的核上面，这些核上面有其他线程在执行，并没有检查该核上执行的线程是否可以移动到空闲核上面或者其他核上的线程移动之后是否可以空出一个核给被schedule_thread占用的核上的线程执行，那么_tx_thread_smp_remap_solution_find接下来就要检查schedule_thread线程映射的核上的线程是否可以移动到其他核上执行。

移动核的循环实现理解不是很直观，简单理解就是，新的线程去占用所有可能的核，检查这些核上原来的线程是不是可以移动到空闲核上，如果找到了一个可以移动到空闲核的线程，那么就退出，新的线程占用该核即可(其他核保持不变)，如果找不到可以移动到空闲核的线程，那么被新线程占用的线程就去占用其他可能的核，其他核上的线程可能存在可以移动的空闲核的线程，递归下去，直到有一个被占用的线程可以移动到空闲核上面去，或者找不到可占用的核了；如果找到了，那么通过移动到空闲核的线程就可以反过来找到线程的移动路径。

4.1、核占用及线程移动过程实例

占用移动过程比较抽象，先以一个例子来演示一下整个过程，例如：线程8为新调度的线程，数组就是核，核下标从1开始，数组方框里面的数值是线程编号，第1个核里面执行的是线程1，当前的线程编号正好与核的编号相同(只是为了简单演示，实际情况线程是随机分布的)，如下图所示:

1：1~3核执行
2：2~4核执行
3：3~5核执行
4：4~6核执行
5：5~7核执行
6：6~8核执行
7：7核执行
8：可以在3~5核执行(线程8为新线程)

1	2	3	4	5	6	7	空闲

步骤1: 线程8占用所有可执行的核3~5，占用如下所示，小括号里面的是原来执行的线程，灰色为被占用的核

1：1~3核执行
2：2~4核执行
3：3~5核执行
4：4~6核执行
5：5~7核执行
6：6~8核执行
7：7核执行
8：可以在3~5核执行(线程8为新线程)

1	2	8(3)	8(4)	8(5)	6	7	空闲

占用核的队列：3、4、5

步骤2: 从队列开始，检查占用核的线程是否可以移动的空闲线程，此时，队列里面第一个核是3，核3出队列，检查线程8占用的核3，看看上面执行的线程是不是可以移动的空闲核上，很显然，线程3不能移动到核8上面执行，那么线程3去占用其他可执行的核，很明显，线程3没有可以占用的核了

步骤3: 从队列开始，检查占用核的线程是否可以移动的空闲线程，此时，队列里面第一个核是4，核4出队列，检查线程8占用的核4，看看上面执行的线程是不是可以移动的空闲核上，很显然，线程4不能移动到核8上面执行，那么线程4去占用其他可执行的核，很明显，线程4只能占用核6，核6入栈

1：1~3核执行
2：2~4核执行
3：3~5核执行
4：4~6核执行
5：5~7核执行
6：6~8核执行
7：7核执行
8：可以在3~5核执行(线程8为新线程)

1	2	8(3)	8(4)	8(5)	4(6)	7	空闲

占用核的队列：5、6

步骤4: 从队列开始，检查占用核的线程是否可以移动的空闲线程，此时，队列里面第一个核是5，核5出队列，检查线程8占用的核5，看看上面执行的线程是不是可以移动的空闲核上，很显然，线程5不能移动到核8上面执行，那么线程5去占用其他可执行的核，很明显，线程4只能占用核7，核7入栈

1：1~3核执行
2：2~4核执行
3：3~5核执行
4：4~6核执行
5：5~7核执行
6：6~8核执行
7：7核执行
8：可以在3~5核执行(线程8为新线程)

1	2	8(3)	8(4)	8(5)	4(6)	5(7)	空闲

占用核的队列：6、7

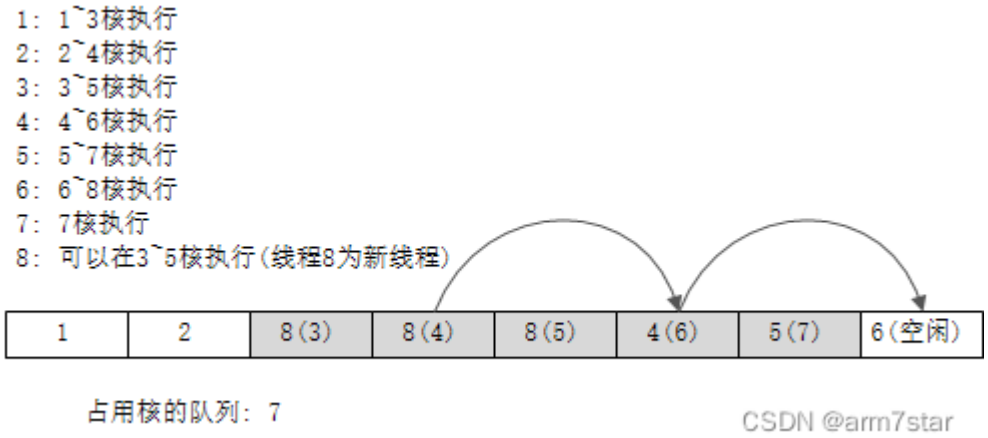
步骤5: 从队列开始，检查占用核的线程是否可以移动的空闲线程，此时，队列里面第一个核是6，核6出队列，检查线程8占用的核6，看看上面执行的线程是不是可以移动的空闲核上，很显然，线程6可以移动到空闲核上面，那么查找结束，记录线程6为最后一个线程

1：1~3核执行
2：2~4核执行
3：3~5核执行
4：4~6核执行
5：5~7核执行
6：6~8核执行
7：7核执行
8：可以在3~5核执行(线程8为新线程)

1	2	8(3)	8(4)	8(5)	4(6)	5(7)	6(空闲)

占用核的队列：7

步骤6: 从最后一个移动的线程反过来查找移动路径，如下图所示，线程6移动到核8，线程6原来在核6上面执行，那么就找哪个线程应该移动到核6，现在看到的是核6被线程4占用，那么线程4就应该移动到核6，接着找哪个线程应该移动到线程4原来所在的核4，核4线程是线程8占用，那么线程8就应该移动到核4上面执行，因为线程8就是新的线程，那么就移动结束了，没有移动的位置就不用移动了，例如核3被8占用，但是核3不需要移动，最终核3还是执行线程3，



4.2、占用核的过程代码

占用核的整个过程就是找到移动到空闲核的那个线程，实现代码如下：

```
1226     /* Loop to evaluate the potential thread mappings, against what is already
    mapped.  */
2227     do
3228     {
4229
5230         /* Pickup the next entry.  */
6231         core = core_queue[queue_first]; // 取第一个占用的核
7232
8233         /* Move to next slot.  */
9234         queue_first++; // 队列头指针后移(第一个核已经出队列了，指向队列里面的下
    一个元素)
10235
11236         /* Retrieve the thread from the current mapping.  */
12237         thread_ptr =  _tx_thread_smp_schedule_list[core]; // 获取被占用的core核
    上正在执行的线程
13238
14239         /* Determine if there is a thread currently mapped to this core.  */
15240         if (thread_ptr != TX_NULL) // core核上有线程正在执行，正在执行的线程为
    thread_ptr，那么新线程放到这个core上就需要将thread_ptr线程移动到其他核上
16241         {
17242
18243             /* Determine the cores available for this thread.  */
19244             thread_possible_cores =  thread_ptr -> tx_thread_smp_cores_allowed;
    // thread_ptr线程允许运行的核(thread_ptr线程绑定的核)
20245             thread_possible_cores =  test_possible_cores &
    thread_possible_cores; // 所有正在执行的线程可以移动的核test_possible_cores并上
    thread_ptr线程绑定的核就得到thread_ptr线程当前可以移动过去的核
21246
22247             /* Are there any possible cores for this thread?  */
23248             if (thread_possible_cores != ((ULONG) 0)) // 如果有核可以移动过去，
    那么就移动该线程(移动到空闲核或者其他线程的核)
24249             {
25250
26251                 /* Determine if there are cores available for this thread.  */
27252                 if ((thread_possible_cores & available_cores) != ((ULONG) 0))
    // thread_ptr线程可以移动到空闲核上面，那么就移动到空闲核上面
```

```
28. 1253                {

29. 1254

30. 1255                    /* Yes, remember the final thread and cores that are valid
    for this thread.  */

31. 1256                    last_thread_cores =  thread_possible_cores &
    available_cores; // 记录thread_ptr线程可移动过去的空闲核(可能有多个空闲核，最终后面
    会取最小的核)

32. 1257                    last_thread =        thread_ptr; // 记录最后一个移动的线程
    (通过最后一个线程来反向查找前面的线程)

33. 1258

34. 1259                    /* We are done - get out of the loop!  */

35. 1260                    break; // 找到合适的映射方案，退出循环

36. 1261                }

37. 1262                else // 不能移动到空闲核上面，那么移动thread_ptr线程到其他可能
    的核上面

38. 1263                {

39. 1264

40. 1265                    /* Remove cores that will be added to the list.  */

41. 1266                    test_possible_cores =  test_possible_cores & ~
    (thread_possible_cores); // thread_ptr线程会占用所有可能的核，那么
    test_possible_cores就要清除掉thread_ptr线程所占用的核，就剩下其他线程可占用的核；这
    里就是减掉占用的核

42. 1267

43. 1268                    /* Loop to add this thread to the potential mapping list.
    */

44. 1269                    do // thread_ptr线程占用所有可能的核(可能移动过去的核)

45. 1270                    {

46. 1271

47. 1272                        /* Calculate the core.  */

48. 1273                        test_cores =  thread_possible_cores;

49. 1274                        TX_LOWEST_SET_BIT_CALCULATE(test_cores, core) // 所有可
    能的核取最小的核，优先移动到最小的核上面去

50. 1275

51. 1276                        /* Clear this core.  */

52. 1277                        thread_possible_cores =  thread_possible_cores & ~
    (((ULONG) 1) << core); // 清除最小的核(最小的核已经取出来了)

53. 1278
```

```
1279                            /* Remember this thread for remapping.  */
1280                            thread_remap_list[core] =  thread_ptr; // thread_ptr线程占用核core
1281
1282                            /* Remember this core.  */
1283                            core_queue[queue_last] =  core; // 占用的核core入队列，do ... while循环依次检查占用的核，检查完的核出队列，还没检查的核入队列
1284
1285                            /* Move to next slot.  */
1286                            queue_last++; // 队列尾指针移动到下一个元素(下一个存储地址)
1287
1288                        } while (thread_possible_cores != ((ULONG) 0)); // 还有可能的核，那么继续去占用这些可能的核
1289                    }
1290                }
1291            }
1292        } while (queue_first != queue_last); // 所有占用的核检查完了，那么都没找到可移动到空闲核的，就不存在可行的移动方案
```

4.3、反向查找移动路径

通过最后一个移动到空闲核的线程，反向查找前一个线程，直至找到新的调度线程，实现代码如下：

```
1294     /* Was a remapping solution found?  */
1295     if (last_thread != TX_NULL) // last_thread记录移动到空闲核的线程，如果不为空，那么就找到了，否则没有找到
1296     {
1297
1298         /* Pickup the core of the last thread to remap.  */
1299         core =  last_thread -> tx_thread_smp_core_mapped; // 获取last_thread线程所映射的核(移动之前所在的核)
1300
1301         /* Pickup the thread from the remapping list.  */
1302         thread_ptr =  thread_remap_list[core]; // 获取占用last_thread线程执行核的线程(哪个线程需要移动到last_thread线程之前所在的核)
1303
1304         /* Loop until we arrive at the thread we have been trying to map.  */
1305         while (thread_ptr != schedule_thread) // 不是新的schedule_thread线程，除了要将thread_ptr线程移动到last_thread线程之前所在核之外，还要知道哪一个线程移动到thread_ptr线程之前所在核，循环直到反向找到新的schedule_thread线程
1306         {
1307
1308             /* Move this thread in the schedule list.  */
1309             _tx_thread_smp_schedule_list[core] =  thread_ptr; // thread_ptr线程移动到核core上执行
1310
1311             /* Remember the previous core.  */
1312             previous_core =  core; // 记录thread_ptr线程执行的核，后面再更新到thread_ptr线程
1313
1314             /* Pickup the core of thread to remap.  */
1315             core =  thread_ptr -> tx_thread_smp_core_mapped; // 获取thread_ptr线程所映射的核(移动之前所在的核)
1316
1317             /* Save the new core mapping for this thread.  */
1318             thread_ptr -> tx_thread_smp_core_mapped =  previous_core; // 更新thread_ptr线程所映射的核
1319
1320             /* Move the next thread.  */
```

```
28. 1321             thread_ptr =  thread_remap_list[core]; // 获取映射到core的线程(thread_ptr移动到该核执行)
29. 1322         }
30. 1323
31. 1324         /* Save the remaining thread in the updated schedule list.  */
32. 1325         _tx_thread_smp_schedule_list[core] =  thread_ptr; // thread_ptr == schedule_thread上面的循环结束，因此这里的thread_ptr就是schedule_thread，让schedule_thread线程在核core上面执行
33. 1326
34. 1327         /* Update this thread's core mapping.  */
35. 1328         thread_ptr -> tx_thread_smp_core_mapped =  core; // 记录schedule_thread线程映射的核
36. 1329
37. 1330         /* Finally, setup the last thread in the remapping solution.  */
38. 1331         test_cores =  last_thread_cores; // 最后一个移动线程可移动的核
39. 1332         TX_LOWEST_SET_BIT_CALCULATE(test_cores, core) // 选最小的一个可以移动过去的核作为最后一个移动线程的执行核
40. 1333
41. 1334         /* Setup the last thread.  */
42. 1335         _tx_thread_smp_schedule_list[core] =     last_thread; // 将最后一个移动的线程放到核core上执行
43. 1336
44. 1337         /* Remember the core mapping for this thread.  */
45. 1338         last_thread -> tx_thread_smp_core_mapped =  core; // 更新last_thread线程所映射的核
46. 1339     }
47. 1340     else // 没有找到合适的移动方案，返回TX_THREAD_SMP_MAX_CORES(无效的核，上一级函数就会尝试获取下一个就绪线程，看看是否可以移动到某个核上执行)
48. 1341     {
49. 1342
50. 1343         /* Set core to the maximum value in order to signal a remapping solution was not found.  */
51. 1344         core =  ((UINT) TX_THREAD_SMP_MAX_CORES);
52. 1345     }
```

5、总结

整个过程就是先把能占的核先占了，如果被占的核上的线程能移动到空闲核上，那么就移动到空闲核上面去，如果不能，那就递归占能占的核。