

三头六臂——一起做几个多线程的例程

前面介绍了如何创建和删除线程（记住我们不建议删除线程）。这节课我们就多创建几个线程来好好分析一下多线程之间的调度。

请打开 sample3 工程，我们看一下代码。

1.1.1 创建三个线程

既然是三头六臂，那我们就建立三个线程吧。

```
100: int main(void)
101: {
102:     rt_thread_t tid;
103:     uint16_t pinStatus=0;
104:
105:     /* set LED2 pin mode to output */
106:     rt_pin_mode(LED2_PIN, PIN_MODE_OUTPUT);
107:     /* set LED3 pin mode to output */
108:     rt_pin_mode(LED3_PIN, PIN_MODE_OUTPUT);
109:     /* set LED4 pin mode to output */
110:     rt_pin_mode(LED4_PIN, PIN_MODE_OUTPUT);
111:
112:     rt_scheduler_sethook(printSchedule);
113:
114:     tid = rt_thread_create("LED2Thread", LED2Thread, RT_NULL, LED2_STACK_SIZE, LED2_PRIO, LED2_TICKS);
115:     RT_ASSERT(tid != RT_NULL);
116:     rt_thread_startup(tid);
117:
118:     tid = rt_thread_create("LED3Thread", LED3Thread, RT_NULL, LED3_STACK_SIZE, LED3_PRIO, LED3_TICKS);
119:     RT_ASSERT(tid != RT_NULL);
120:     rt_thread_startup(tid);
121:
122:     while (1)
123:     {
124:         pinStatus=rt_pin_read(LED4_PIN);
125:         if(pinStatus)
126:             rt_pin_write(LED4_PIN, PIN_LOW);
127:         else
128:             rt_pin_write(LED4_PIN, PIN_HIGH);
129:         #if HARD_DELAY==1
130:             _delay_us(SPEED*1000);
131:         #else
132:             rt_thread_mdelay(SPEED);
133:         #endif
134:     }
135: } « end main »
```

图 4.1: 主线程函数

从主线程函数里我们看到，分别创建了两个线程，“LED2Thread()”负责 LED2 的闪烁功能，“LED3Thread()”负责 LED3 的闪烁功能。主线程自己负责

LED4 的闪烁功能。三个线程的优先级全部设定为同一个级别，因此从理论上说他们三个之间不存在谁打断谁的情况。同时，我们把主线程、LED2 和 LED3 的 TICKS 设置为 1000。也就是大约 1 秒中才调度一次，设置的这么大的原因是为了实验现象比较明显而为之。

```
23: void LED2Thread(void)
24: {
25:     //uint16_t i;
26:     uint32_t Speed = 200;
27:     uint16_t pinStatus=0;
28:     //for(i=0;i<10;i++)
29:     while(1)
30:     {
31:         pinStatus=rt_pin_read(LED2_PIN);
32:         if(pinStatus)
33:             rt_pin_write(LED2_PIN, PIN_LOW);
34:         else
35:             rt_pin_write(LED2_PIN, PIN_HIGH);
36:         rt_thread_mdelay(Speed);
37:     }
38:     rt_kprintf("Goodby~!\n");
39: }
40:
41: //MSH_CMD_EXPORT(LED2Thread, LED2Thread);
42:
43: void LED3Thread(void)
44: {
45:     //uint16_t i;
46:     uint32_t Speed = 200;
47:     uint16_t pinStatus=0;
48:     //for(i=0;i<10;i++)
49:     while(1)
50:     {
51:         pinStatus=rt_pin_read(LED3_PIN);
52:         if(pinStatus)
53:             rt_pin_write(LED3_PIN, PIN_LOW);
54:         else
55:             rt_pin_write(LED3_PIN, PIN_HIGH);
56:         rt_thread_mdelay(Speed);
57:     }
58:     rt_kprintf("Goodby~!\n");
59: }
60: }
```

图 4.2: LED2 和 LED3 线程函数

在继续讲下去前我们再回顾一下 OS 线程调度的知识。所谓的线程调度其实就是 OS 根据线程（其实就是函数）的优先级和时间片来定时运行函数。在前面提到过，内核会把所有线程对象链接在一个链表里。OS 会根据系统时钟，也就是 Tick 到链表里检查线程（函数）是否该被运行。比如，一个函数调用了

“rt_thread_mdelay(x)” 函数，那么这个线程就释放了 CPU 的使用权，并告知 OS 请在 x 毫秒时间后再来继续执行我的代码。于是这个线程就被挂起，OS 就会在链表里找，看看有没有其他的线程可以运行，如果都没有就会运行系统自己创建的空闲线程。空闲线程里会看看系统中有没有可以回收的资源，如果有，就会回收资源并释放被这些资源占用的内存。同时，系统的低功耗进入程序一般也是由空闲线程实现的，后面我们还会解释。线程的状态和状态切换图一般如下：

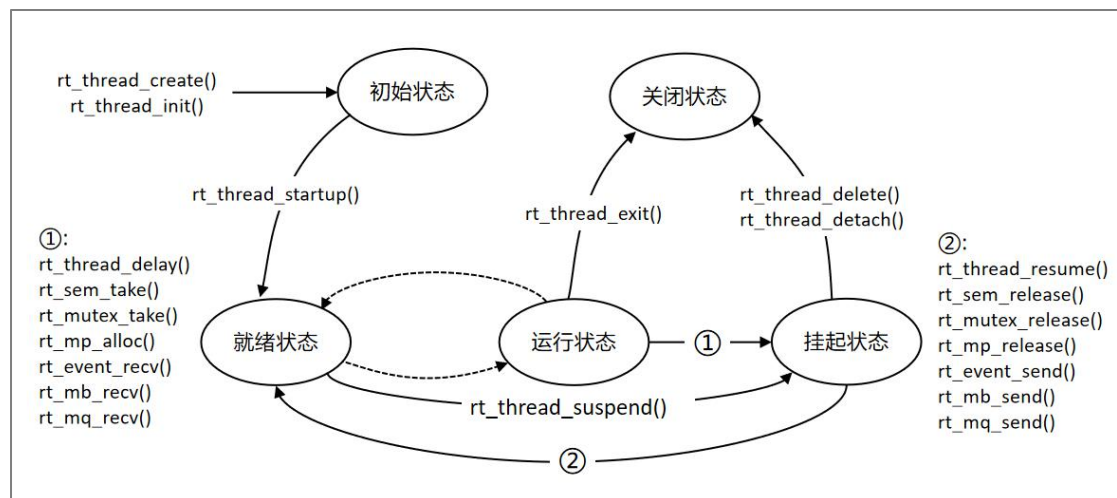


图 4.3:线程的状态和切换

好，让我们编译、下载和观察一下。如果顺利，各位会发现这个程序让三个 LED 灯几乎同时在闪烁。但是，我们从代码的顺序上看，貌似是 LED4 先亮，然后 LED2 再亮，最后是 LED3 亮。因为三个线程是相同优先级的，其实他们之间是不存在打断现象的。所以应该按照顺序来闪烁，就像以前的跑马灯一样。那为何现在是三个灯同时闪烁呢？主要的问题在于那句 “rt_thread_mdelay()”，这是一句 RTT 的线程延时函数，这个函数一旦被调用，当前的线程就会立即释放 CPU 的使用权，让 OS 调度其他的线程来执行，直到延时时间到了后，才会继续执行。这种主动让出 CPU 的函数，在 RTOS 的编程中应该尽量使用，以提高

系统利用率。还是这个例子，我们把“rt_thread_mdelay()”改成“_delay_us()”来实现，一切就不同了。

```
32: #if HARD_DELAY==1
33: static void _delay_us(uint32_t us)
34: {
35:     volatile uint32_t len;
36:     for (; us > 0; us--)
37:         for (len = 0; len < 20; len++);
38: }
39: #endif
```

图 4.4: _delay_us()函数

我们先看一下代码中的“_delay_us()”函数的实现，其实就是一个二阶的循环体，相当于让 CPU 不断运行这个循环体来消耗时间。这个函数就模拟了前面提过的，始终占用 CPU 资源，而不主动让出 CPU。同样，也是延时大约 200mS，我们看看运行结果有何不同。

```
10:
11: #include <rtthread.h>
12: #include <rtdevice.h>
13: #include "board.h"
14: #include "drv_gpio.h"
15:
16: #define HARD_DELAY 1
17: #define SPEED 200
18: #define LED2_PRIO 10
19: #define LED3_PRIO 10
20: #define LED2_STACK_SIZE 256
21: #define LED3_STACK_SIZE 256
22: #define LED2_TICKS 1000
23: #define LED3_TICKS 1000
24:
25: /* defined the LED2 pin: PD13 */
26: #define LED2_PIN GET_PIN(D, 13)
27: /* defined the LED3 pin: PD14 */
28: #define LED3_PIN GET_PIN(D, 14)
29: /* defined the LED4 pin: PD15 */
30: #define LED4_PIN GET_PIN(D, 15)
31:
32: #if HARD_DELAY==1
33: static void _delay_us(uint32_t us)
34: {
```

图 4.4: HARD_DELAY 宏定义

在编译、下载前请确认“HARD_DELAY”宏定义被设置成了1。为了调试方便，我添加了这个宏定义开关，用来控制延时函数到底是用“硬”延时还是用OS的“软”延时。好了，请观察LED灯的闪烁情况。我们发现三个LED灯开始按照先后次序闪烁，每个灯大约闪烁1秒中后才会切换到后面一个灯。这是因为，线程完全占据了CPU的时间，知道自己的1秒钟全部用完才“罢休”，三个线程的优先级又相同，所以自然就只能按照顺序来工作了。这种工作方式效率低下，其实和没有OS差不多。因此在实际工作中要避免。

在我们回到正常的延时操作之前，我们先来修改线程的优先级，试试看。主线程在创建时的优先级被定义为10,我们把LED2和LED3的优先级改成8和9。注意，在RTT中数字越小，优先级越高。

```
16 #define HARD_DELAY 1
17 #define SPEED 200
18 #define LED2_PRIO 8
19 #define LED3_PRIO 9
20 #define LED2_STACK_SIZE 256
21 #define LED3_STACK_SIZE 256
22 #define LED2_TICKS 1000
23 #define LED3_TICKS 1000
24
```

图 4.4:线程优先级的宏定义

修改后，让我们继续再次编译、下载和执行。观察运行结果，你会发现除了LED2在闪烁其他两个LED都不会动作了。原因很简单——那位LED2的线程优先级最高，导致其一直抢占LED3和LED4的线程。最后就表现为只有LED2在工作了。这很好的验证了抢占式调度的机制。

此时，我们再将延时函数用比较正常的“rt_thread_mdelay()”来实验一下。请将“HARD_DELAY”宏定义设置为0，然后重新编译和下载，观察现象。你

会惊奇的发现，其运行效果和当初一样，三个 LED 灯几乎被同时操作。道理也非常简单，因为每个线程都会利用系统提供的延时函数来让出 CPU 的执行时间，即使优先级不同，但是低优先级的线程也有充足的机会去运行程序，于是看起来就“一团和气”。正所谓“与人方便、于己方便”啊~！

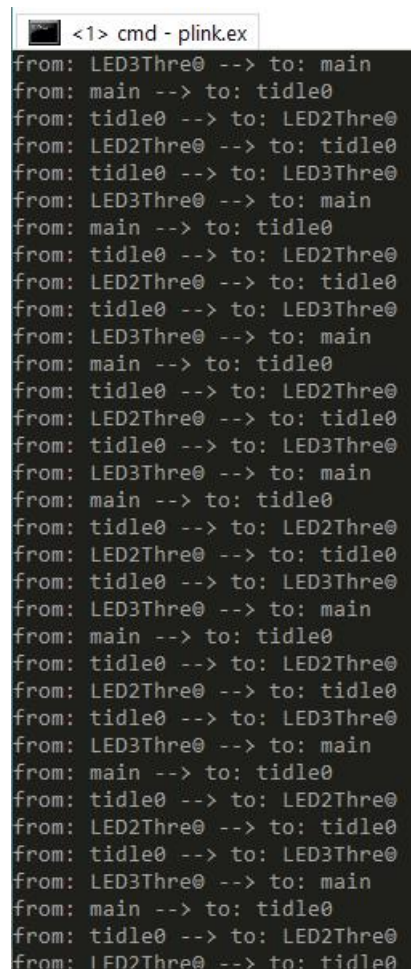
1.1.2 如何观察线程调度情况

前面我们用 LED 灯的闪烁情况来演示了 OS 对三个线程的调度情况，在实际工作中我们未必能这么直观的来观察线程的调度情况，而且线程也可能多于三个。RTT 提供了一种称之为“钩子函数(hook)”的机制来帮助我们调试。所谓“钩子函数”其实就是一个回调函数，当 OS 运行到一些关键点时，可以调用回调函数，只要用户设置了这个回调函数，那么就能显式的看到 OS 执行的一些关键过程。和线程调度相关的钩子函数可以通过 OS 提供的 API 函数“rt_scheduler_sethook()”来设定。具体的可以看代码。

```
95: void printSchdule(struct rt_thread *from, struct rt_thread *to)
96: {
97:     rt_kprintf("from: %s --> to: %s \n", from->name, to->name);
98: }
99:
100: int main(void)
101: {
102:     rt_thread_t tid;
103:     uint16_t pinStatus=0;
104:
105:     /* set LED2 pin mode to output */
106:     rt_pin_mode(LED2_PIN, PIN_MODE_OUTPUT);
107:     /* set LED3 pin mode to output */
108:     rt_pin_mode(LED3_PIN, PIN_MODE_OUTPUT);
109:     /* set LED4 pin mode to output */
110:     rt_pin_mode(LED4_PIN, PIN_MODE_OUTPUT);
111:
112:     rt_scheduler_sethook(printSchdule);
113:
114:     tid = rt_thread_create("LED2Thread", LED2Thread, RT_NULL, LED2_STACK_SIZE, LED2_PRIO, LED2_TICKS);
115:     RT_ASSERT(tid != RT_NULL);
116: }
```

图 4.5:线程调度钩子函数

通过 “rt_scheduler_sethook()” 来注册一个用户的钩子函数，这里我命名成 “printSchdule()”。有两个参数，一个当前的线程的线程控制块指针，另一个是将被调度的线程控制块指针。函数中只有一句话，就是将现在线程的名字和将要调度的线程的名字打印出来。我们编译并运行一下，看看效果。请连接 putty 终端，然后我们就能看到其线程切换的过程。



```
<1> cmd - plink.exe
from: LED3Thre@ --> to: main
from: main --> to: tidle0
from: tidle0 --> to: LED2Thre@
from: LED2Thre@ --> to: tidle0
from: tidle0 --> to: LED3Thre@
from: LED3Thre@ --> to: main
from: main --> to: tidle0
from: tidle0 --> to: LED2Thre@
from: LED2Thre@ --> to: tidle0
from: tidle0 --> to: LED3Thre@
from: LED3Thre@ --> to: main
from: main --> to: tidle0
from: tidle0 --> to: LED2Thre@
from: LED2Thre@ --> to: tidle0
from: tidle0 --> to: LED3Thre@
from: LED3Thre@ --> to: main
from: main --> to: tidle0
from: tidle0 --> to: LED2Thre@
from: LED2Thre@ --> to: tidle0
from: tidle0 --> to: LED3Thre@
from: LED3Thre@ --> to: main
from: main --> to: tidle0
from: tidle0 --> to: LED2Thre@
from: LED2Thre@ --> to: tidle0
from: tidle0 --> to: LED3Thre@
from: LED3Thre@ --> to: main
from: main --> to: tidle0
from: tidle0 --> to: LED2Thre@
from: LED2Thre@ --> to: tidle0
```

图 4.6:线程调度钩子函数打印出来的信息

我们能看到，线程的调度顺序基本上是 “main→空闲→LED2 线程→空闲→LED3 线程→main……”。请结合代码来观察中程序，我觉得可以很方便的梳理出逻辑。

钩子函数虽然用起来非常方便，但是不停的从钩子函数里打印字符串出来，非常消耗资源，而且持续“霸屏”，也挺烦人的。我们可以考虑另外一个办法，当然这个办法需要些设备来配合，比如示波器。我们现在有三个线程，正好也有三个 LED 灯，我们可以利用驱动这三个 LED 灯的 GPIO 来显示波形。我们需要修改一下代码，请打开 sample4 工程。在这个工程里，我的核心思想是，线程如果正常工作，那么就把 GPIO 拉低，如果线程被调度，那么就把 GPIO 拉高。这样，我们通过示波器就能观察线程的调度情况。

```
87: int main(void)
88: {
89:     rt_thread_t tid;
90:     uint16_t pinStatus=0;
91:
92:     /* set LED2 pin mode to output */
93:     rt_pin_mode(LED2_PIN, PIN_MODE_OUTPUT);
94:     /* set LED3 pin mode to output */
95:     rt_pin_mode(LED3_PIN, PIN_MODE_OUTPUT);
96:     /* set LED4 pin mode to output */
97:     rt_pin_mode(LED4_PIN, PIN_MODE_OUTPUT);
98:
99:     rt_scheduler_sethook(showSchedule);
100:
101:     tid = rt_thread_create("LED2Thread", LED2Thread, RT_NULL, LED2_STACK_SIZE, LED2_PRIO, LED2_TICKS);
102:     RT_ASSERT(tid != RT_NULL);
103:     rt_thread_startup(tid);
104:
105:     tid = rt_thread_create("LED3Thread", LED3Thread, RT_NULL, LED3_STACK_SIZE, LED3_PRIO, LED3_TICKS);
106:     RT_ASSERT(tid != RT_NULL);
107:     rt_thread_startup(tid);
108:
109:     while (1)
110:     {
111:         //pinStatus=rt_pin_read(LED4_PIN);
112:         //if(pinStatus)
113:             rt_pin_write(LED4_PIN, PIN_LOW);
114:         //else
115:             // rt_pin_write(LED4_PIN, PIN_HIGH);
116:
117:         rt_thread_mdelay(SPEED);
118:     }
119: } « end main »
```

图 4.7:sample4 中的 main 函数


```

79: void showSchedule(struct rt_thread *from, struct rt_thread *to)
80: {
81:     //rt_kprintf("from: %s --> to: %s \n", from->name, to->name);
82:     rt_pin_write(LED2_PIN, PIN_HIGH);
83:     rt_pin_write(LED3_PIN, PIN_HIGH);
84:     rt_pin_write(LED4_PIN, PIN_HIGH);
85: }
86:
87: int main(void)
88: {
89:     rt_thread_t tid;
90:     uint16_t pinStatus=0;
91:
92:     /* set LED2 pin mode to output */
93:     rt_pin_mode(LED2_PIN, PIN_MODE_OUTPUT);
94:     /* set LED3 pin mode to output */
95:     rt_pin_mode(LED3_PIN, PIN_MODE_OUTPUT);
96:     /* set LED4 pin mode to output */
97:     rt_pin_mode(LED4_PIN, PIN_MODE_OUTPUT);
98:
99:     rt_scheduler_sethook(showSchedule);
100:
101:     tid = rt_thread_create("LED2Thread", LED2Thread, RT_NULL, LED2_STACK_SIZE, LED2_PRIO, LED2_TICKS);
102:     RT_ASSERT(tid != RT_NULL);

```

图 4.8:sample4 中的 showSchedule 函数

现在我们可以看看效果，当然前提是手头要有示波器。如果手头没有，那么就看看下图。

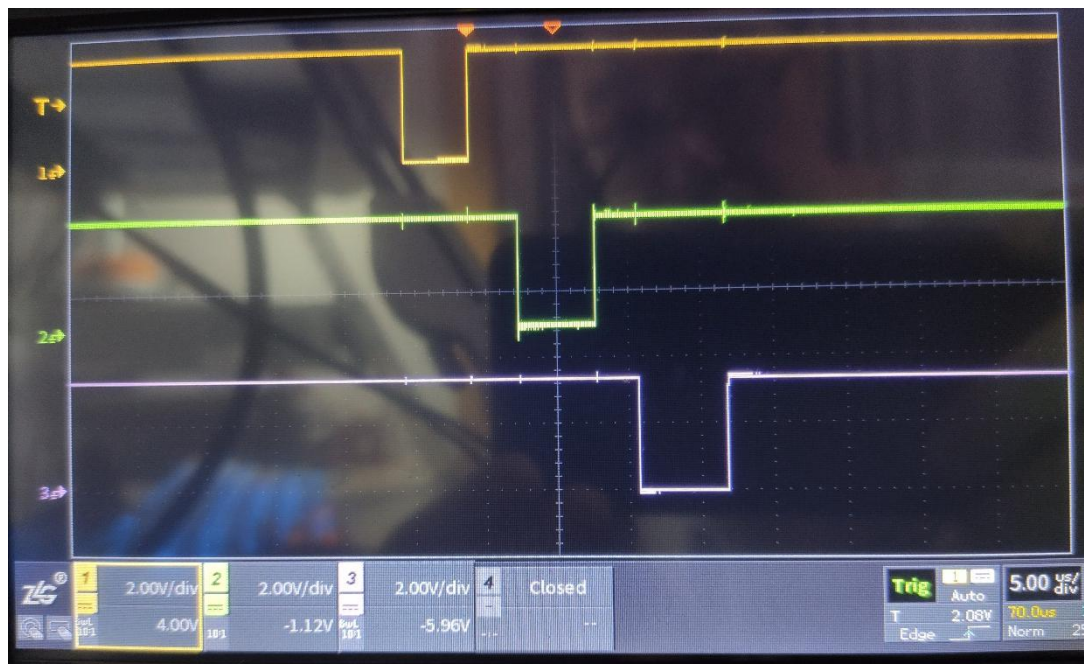


图 4.9:sample4 执行效果

从上图我们能清晰的看出三个线程的调度先后，编号 1 是 LED2 线程，编号 2 是 LED3 的线程，编号 3 是 LED4(也就是主线程)。他们依次被调度，然后因

为调用了系统延迟函数，而放弃 CPU 使用，于是都不工作，直到延时时间到了。这种调试方式实时性非常好，也能精确测定线程的执行时间。不过对于硬件要求比较高，会占用不少 GPIO，还需要用到其他测试设备。在实际工作中，还需要大家根据实际情况来选择合适的测量方式。