

# RT-Thread 内核实验 1 任务的基本管理

## 实验目的：

- ◇ 理解 RTOS 中任务管理的基本原理，了解任务的基本状态及其变迁过程；
- ◇ 掌握 RT-Thread 任务管理子系统任务中的任务创建，启动，延时机制
- ◇ 掌握 RT-Thread 任务管理子系统中静态任务与动态任务创建的区别

## 实验设计：

为了体现任务的状态切换过程，本实验设计了两个线程，分别是 thread1，thread2，此外，系统中还有另外一个空闲任务，当没有其他任务运行时，系统自动调度空闲任务并投入运行。



## 实验流程：

- 1) 系统进入应用程序入口
- 2) 初始化静态线程 thread1，线程的入口是 thread1\_entry,参数是 RT\_NULL，线程栈是 thread1\_stack,优先级是 30，时间片是 10 个 OS Tick
- 3) 启动线程 thread1
- 4) 创建动态线程 thread2，线程的入口是 thread2\_entry,，参数是 RT\_NULL，栈空间是 512，优先级是 31，时间片是 25 个 OS Tick
- 5) 启动线程 thread2
- 6) [1]系统首先调度 thread1 投入运行，打印第 0 次运行的信息，然后通过延时函数将自己挂起 100 个 OS Tick，系统转入调度 thread2 投入运行
- 7) [2]Thread2 打印第 0 次运行信息，然后通过延时函数将自己挂起 50 个 OS Tick
- 8) [3]系统中无任务运行，转入调度空闲任务
- 9) [4]50 个 OS Tick 时间后，Thread2 被唤醒，打印第 1 次运行的信息，继续通过延时函数将自己挂起 50 个 OS Tick
- 10) [5]系统中无任务运行，系统转入调度空闲任务，运行 50 个 OS Tick

100个OS Tick约为1S  
rt\_thread\_delay(1);为10ms

- 11) [6]Thread1 被唤醒，打印第 1 次运行信息，继续挂起 100 个 OS Tick
- 12) [7]Thread2 被唤醒，打印第 2 次运行的信息，继续挂起 50 个 OS Tick
- 13) [8]系统中无任务运行，系统转入调度空闲任务,运行 50 个 OS Tick
- 14) [9]Thread2 被唤醒，打印第 3 次运行的信息，继续挂起 50 个 OS Tick
- 15) [10]系统中无任务运行，系统转入调度空闲任务，运行 50 个 OS Tick
- 16) 循环上述过程

## 源程序说明：

```
/* 线程1的对象和运行时用到的栈 */
static struct rt_thread thread1;
ALIGN(4)
static rt_uint8_t thread1_stack[512];

/* 线程1入口 */
void thread1_entry(void* parameter)
{
    int i;

    while (1)
    {
        for (i = 0; i < 10; i++)
        {
            rt_kprintf("%d\n", i);

            /* 延时100个OS Tick */
            rt_thread_delay(100);
        }
    }
}

/* 线程2入口 */
void thread2_entry(void* parameter)
{
    int count = 0;
    while (1)
    {
        rt_kprintf("Thread2 count:%d\n", ++count);

        /* 延时50个OS Tick */
        rt_thread_delay(50);
    }
}

/* 用户应用程序入口 */
int rt_application_init()
{
    rt_thread_t thread2_ptr;
    rt_err_t result;
```

```

/* 初始化线程1 */
/* 线程的入口是thread1_entry, 参数是RT_NULL
 * 线程栈是thread1_stack
 * 优先级是200, 时间片是10个OS Tick
 */
result = rt_thread_init(&thread1,
    "thread1",
    thread1_entry, RT_NULL,
    &thread1_stack[0], sizeof(thread1_stack),
    30, 10);

/* 启动线程 */
if (result == RT_EOK) rt_thread_startup(&thread1);

/* 创建线程2 */
/* 线程的入口是thread2_entry, 参数是RT_NULL
 * 栈空间是512, 优先级是250, 时间片是25个OS Tick
 */
thread2_ptr = rt_thread_create("thread2",
    thread2_entry, RT_NULL,
    512, 31, 25);

/* 启动线程 */
if (thread2_ptr != RT_NULL) rt_thread_startup(thread2_ptr);

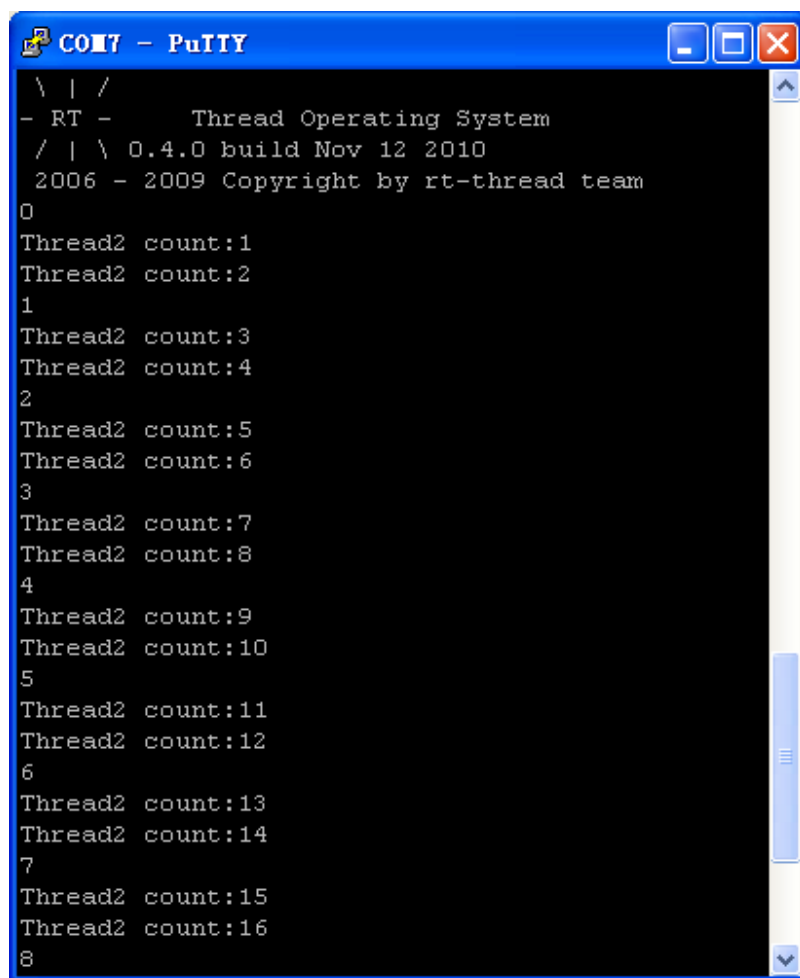
return 0;
}

```

初始化线程，与创建线程功能相同。最后再启动线程

## 输出信息：

运行程序，通过观察串口输出，就可以观察到任务的运行和切换情况了。



```
\ | /
- RT -      Thread Operating System
/ | \ 0.4.0 build Nov 12 2010
2006 - 2009 Copyright by rt-thread team
0
Thread2 count:1
Thread2 count:2
1
Thread2 count:3
Thread2 count:4
2
Thread2 count:5
Thread2 count:6
3
Thread2 count:7
Thread2 count:8
4
Thread2 count:9
Thread2 count:10
5
Thread2 count:11
Thread2 count:12
6
Thread2 count:13
Thread2 count:14
7
Thread2 count:15
Thread2 count:16
8
```

## 断点设置

使用 MDK 调试工具在程序中设置一些合理断点来运行，可以清晰地看到线程运行和切换的完整过程。

```

26
27  /* 线程1入口 */
28  void thread1_entry(void* parameter)
29  {
30      int i;
31
32      while (1)
33      {
34          for (i = 0; i < 10; i ++)
35          {
36              rt_kprintf("%d\n", i);
37
38              /* 延时100个OS Tick */
39              rt_thread_delay(100);
40          }
41      }
42  }
43
44  /* 线程2入口 */
45  void thread2_entry(void* parameter)
46  {
47      int count = 0;
48      while (1)
49      {
50          rt_kprintf("Thread2 count:%d\n", ++count);
51

```

## TIPS:动态线程和静态线程

RT-Thread 中支持静态和动态两种定义方式。用线程来举例的话，`rt_thread_init` 对应静态定义方式，`rt_thread_create` 对应动态定义方式。

使用静态定义方式时，必须先定义静态的线程控制块，并且定义好堆栈空间，然后调用 `rt_thread_init` 来完成线程的初始化工作。采用这种方式，线程控制块和堆栈占用的内存会放在 RW 段，这段空间在编译时就已经确定，它不是可以动态分配的，所以不能被释放，而只能使用 `rt_thread_detach` 函数将该线程控制块从对象管理器中脱离。

使用动态定义方式 `rt_thread_create` 时，RT-Thread 会动态申请线程控制块和堆栈空间。在编译时，编译器是不会感知到这段空间的，只有在程序运行时，RT-Thread 才会从系统堆中申请分配这段内存空间，当不需要使用该线程时，调用 `rt_thread_delete` 函数就会将这段申请的内存空间重新释放到内存堆中。

这两种方式各有利弊，静态定义方式会占用 RW/ZI 空间，但是不需要动态分配内存，运行时效率较高，实时性较好。动态方式不会占用额外的 RW/ZI 空间，占用空间小，但是运行时需要动态分配内存，效率没有静态方式高。

总的来说，这两种方式就是空间和时间效率的平衡，可以根据实际环境需求选择采用具体的分配方式。