线程抢占导致临界区问题

RealTouch 评估板 RT-Thread 入门文档

版本号：1.0.0

日期：2012/8/12

修订记录

日期	作者	修订历史
2012/8/12	prife	创建文档

实验目的

- 了解多线程环境中的临界区问题

硬件说明

本实验使用 RT-Thread 官方的 Realtouch 开发板作为实验平台。

涉及到的硬件主要有：

- 串口 3，作为 rt_kprintf 输出，需要连接 JTAG 扩展板

具体请参见《Realtouch 开发板使用手册》

实验原理及程序结构

实验设计

临界资源是指一次仅允许一个线程使用的共享资源。不论是硬件临界资源，还是软件临界资源，多个线程必须互斥地对它们进行访问。

每个线程中访问临界资源的那段代码称为**临界区**(Critical Section)，每次只准许一个线程进入临界区，进入后不允许其他线程进入。

多线程程序的开发方式不同于裸机程序，多个线程在宏观上是并发运行的，因此使用一个共享资源是需要注意，否则就可能出现错误的运行结果。

本实验通过一个简单的共享变量来演示多线程中的临界区问题。

源程序说明

本实验对应 kernel_critial_region。

系统依赖

在 rtconfig.h 中需要开启

- #define RT_USING_HEAP
  此项可选，开启此项可以创建动态线程，如果使用静态线程，则此项不是必要的
- #define RT_USING_CONSOLE

此项必须，本实验使用 rt_kpriintf 向串口打印按键信息，因此需要开启此项

- #define RT_TICK_PER_SECOND 10000

主程序说明

在 applications/application.c 中定义一个全局变量为

定义全局变量

```
static int share_var;
```

在 applications/application.c 中的 int rt_application_init()函数中初始化名为”thread1”的 thread1 的静态线程，如下所示。

创建线程代码

```
    rt_thread_init(&thread1,
                   "thread1",
                   rt_thread_entry1,
                   RT_NULL,
                   &thread1_stack[0],
                   sizeof(thread1_stack),11,5); //线程优先级 11 时间片为 5
    rt_thread_startup(&thread1);
    rt_thread_init(&thread2,
                   "thread2",
                   rt_thread_entry2,
                   RT_NULL,
                   &thread2_stack[0],
                   sizeof(thread2_stack),11,5); //线程优先级 11 时间片为 5
    rt_thread_startup(&thread2);
```

其线程处理函数如下所示，在线程处理函数中，首先调用 rt_kprintf 向串口打印共享变量的值，然后接下来的 for 循环代码来模拟对贡献变量的 share_var 的处理算法，注意这里为了增加这个“算法”的运行时间，使用 for 循环对 share_var 做了 100000 次累加操作。“算法”运行完毕后，再次使用 rt_kprintf 将共享变量的值打印。

线程 1 代码

```
ALIGN(RT_ALIGN_SIZE)            //设置下一句线程栈数组为对齐地址
static char thread1_stack[1024];//设置线程堆栈为 1024Bytes
```

```
struct rt_thread thread1;       //定义静态线程数据结构
static void rt_thread_entry1(void* parameter)
{
    int i;
    share_var = 0;
    rt_kprintf("share_var = %d\n", share_var);

//模拟算法
    for(i=0; i<100000; i++)
    {
        share_var ++;
    }
    rt_kprintf("\t share_var = %d\n", share_var);
}
```

接下来在创建线程 2，代码如下所示，在线程处理函数中，先调用 rt_thread_delay 休眠 1 个系统 tick，然后对共享变量 share_var 的值做一次累加，之后线程处理函数运行完毕。

线程 2 代码

```
ALIGN(RT_ALIGN_SIZE)        //设置下一句线程栈数组为对齐地址
static char thread1_stack[1024];  //设置线程堆栈为 1024Bytes
struct rt_thread thread1;       //定义静态线程数据结构

static void rt_thread_entry2(void* parameter)
{
    rt_thread_delay(1);
    share_var ++;
}
```

编译调试及观察输出信息

编译请参见《RT-Thread 配置开发环境指南》完成编译烧录，参考《Realtouch 开发板使用手册》完成硬件连接，连接扩展板上的串口和 jlink。

运行后可以看到如下信息。

串口输出

```
\ | /
- RT -     Thread Operating System
 / | \     1.1.0 build Aug  6 2012
 2006 - 2012 Copyright by rt-thread team
share_var = 0
        share_var = 100001
```

修改线程 2 中的 rt_thread_delay(1)，为 rt_thread_delay(1000)；再次烧录运行，输入结果如下所示。

串口输出

```
\ | /
- RT -     Thread Operating System
 / | \     1.1.0 build Aug  6 2012
 2006 - 2012 Copyright by rt-thread team
share_var = 0
        share_var = 100000
```

结果分析

为什么会出现上面的结果呢？在 for 循环中我们对 i 做了 100000 次累加，如果没有其他线程的“干预”，那么共享变量的值应该是 100000，现在的输出结果是 100001，这意味这在打共享变量的值发生了变化，这个值是在线程 2 中修改的。

整个程序运行过程中各个线程的状态变化：

rt_application_init 中创建两个线程之后，由于线程 2 的优先级比线程 1 的优先级高，因此线程 2 先运行，其线程处理函数第一句为

rt_thread_delay(1)

这会使得线程 2 被挂起，挂起时间为 1 个 tick，在线程 2 挂起的这段时间中，线程 1 是所有就绪态线程中优先级最高的线程，因此被内核调度运行，在其处理函数中执行，在线程 1 的处理函数执行了一部分代码后，1 个 tick 时间到，线程 1 被唤醒，从而成为所有就绪线程中优先级最高的线程，因此会被立刻调度运行，线程 1 被线程 2 抢占，线程 2 处理函数中对

贡献变量 share_var 做累加操作，接下来线程处理函数执行完毕，线程 1 再次被调度运行，根据程序的运行结果可以看出，此时线程 1 继续执行，但是我们并不知道此时线程 1 大致是从什么地方执行的，从最后的输出结果来看，只能得知此时线程 1 还没有执行到第二条 rt_kprintf 输出语句。最后线程处理函数的最后打印共享变量的值，其值就应该是 100001。

当修改了线程的休眠时间为 1000 个 tick 后，在线程休眠的整个事件内，线程 2 都已经执行完毕，因此最后的输出结果为 100000。

可以看到当共享变量 share_var 在多个线程中公用时，如果缺乏必要的保护错误，最后的输出结果可能与预期的结果完全不同。为了解决这种问题，需要引入线程间通信机制，这就是所谓的 IPC 机制（Inter-Process Communication）。

总结

本实验演示了多任务环境中线程的临界区问题。