线程优先级反转原理

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	bloom5	创建文档

实验目的

- 介绍实时操作系统中的经典问题，优先级反转

硬件说明

本实验使用 RT-Thread 官方的 Realtouch 开发板作为实验平台。涉及到的硬件主要为

- 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

优先级反转是实时操作系统中的经典问题之一，由于多线程共享资源，具有最高优先级的线程被低优先级线程阻塞，反而使中优先级线程先于高优先级线程执行，导致系统故障。

优先级反转的一个典型场景为：系统中存在优先级为 A、B 和 C 的三个线程，优先级 A > B > C，线程 A，B 处于挂起状态，等待某一事件的发生，线程 C 正在运行，此时线程 C 开始使用某一共享资源 S。在使用过程中，线程 A 等待的事件到来，线程 A 转为就绪态，因为它比线程 C 优先级高，所以立即执行。但是当线程 A 要使用共享资源 S 时，由于其正在被线程 C 使用，因此线程 A 被挂起切换到线程 C 运行。如果此时线程 B 等待的事件到来，则线程 B 转为就绪态。由于线程 B 的优先级比线程 C 高，因此线程 B 开始运行，直到其运行完毕，线程 C 才开始运行。只有当线程 C 释放共享资源 S 后，线程 A 才得以执行。在这种情况下，优先级发生了反转，线程 B 先于线程 A 运行。这样便不能保证高优先级线程的响应时间。可见图 1-1

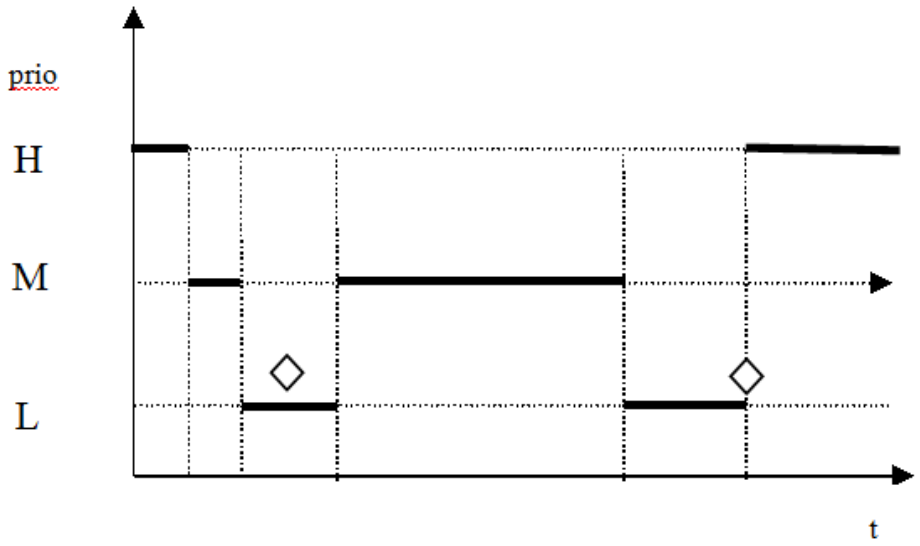

图 1-1

实验设计

本实验的主要设计目的是帮助读者进一步了解优先级反转的实际运行情况，有助于了解优先级反转的原理。

源程序说明

本实验对应 1_kernel_prioinvert

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和动态信号量，如果使用静态线程和静态信号量，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

主程序说明

在 applications/application.c 中定义了三个线程，分别是 t1, t2, worker，以及一个动态信号量 sem

```
t1 = rt_thread_create("t1",
            thread1_entry, RT_NULL,
```

```
            512, 8, 10);
    if (t1 != RT_NULL)
            rt_thread_startup(t1);

 t2 = rt_thread_create("t2",
            thread2_entry, RT_NULL,
            512, 6, 10);
    if (t2 != RT_NULL)
            rt_thread_startup(t2);

 worker = rt_thread_create("worker",
            worker_thread_entry, RT_NULL,
            512, 7, 10);
    if (worker != RT_NULL)
            rt_thread_startup(worker);

 sem = rt_sem_create("sem", 1, RT_IPC_FLAG_PRIO);
    if (sem == RT_NULL)
    {
            return 0;
    }
```

下面的代码是三个线程的入口程序，

```
 static void thread1_entry(void* parameter)
 {
    rt_err_t result;

    result = rt_sem_take(sem, RT_WAITING_FOREVER);

    for(t1_count = 0; t1_count < 10; t1_count ++)
    {
            rt_kprintf("thread1: got semaphore, count: %d\n",
t1_count);
            rt_thread_delay(RT_TICK_PER_SECOND);
    }

    rt_kprintf("thread1: release semaphore\n");
```

```
      rt_sem_release(sem);
  }

  static void thread2_entry(void* parameter)
  {
      rt_err_t result;

      while (1)
      {
              result = rt_sem_take(sem, RT_WAITING_FOREVER);
              rt_kprintf("thread2: got semaphore\n");
              if (result != RT_EOK)
              {
                      return;
              }
              rt_kprintf("thread2: release semaphore\n");
              rt_sem_release(sem);

              rt_thread_delay(5);
              result = rt_sem_take(sem, RT_WAITING_FOREVER);
              t2_count ++;
              rt_kprintf("thread2: got semaphore, count: %d\n",
t2_count);
      }
  }

  static void worker_thread_entry(void* parameter)
  {
      rt_thread_delay(5);
      for(worker_count = 0; worker_count < 10; worker_count++)
      {
              rt_kprintf("worker:  count: %d\n", worker_count);
      }
      rt_thread_delay(RT_TICK_PER_SECOND);
  }
```

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：

```
 \ | /
- RT -     Thread Operating System
 / | \     1.1.0 build Aug 10 2012
 2006 - 2012 Copyright by rt-thread team
thread2: got semaphore
thread2: release semaphore
thread1: got semaphore, count: 0
worker:  count: 0
worker:  count: 1
worker:  count: 2
worker:  count: 3
worker:  count: 4
worker:  count: 5
worker:  count: 6
worker:  count: 7
worker:  count: 8
worker:  count: 9
thread1: got semaphore, count: 1
thread1: got semaphore, count: 2
thread1: got semaphore, count: 3
thread1: got semaphore, count: 4
thread1: got semaphore, count: 5
thread1: got semaphore, count: 6
thread1: got semaphore, count: 7
thread1: got semaphore, count: 8
thread1: got semaphore, count: 9
thread1: release semaphore
thread2: got semaphore, count: 1
….
```

结果分析

三个线程的优先级顺序是 thread2 > thread1 > worker，首先 thread2 得到执行，它得到信号量，并且释放，然后延时等待，然后 worker 线程得到处理器控制权开始运行，它也进行了延时操作，然后，thread1 拿到了控制权，并且它申请得到了信号量，接着进行了打印操作，在它打印结束进行延时操作时，由于 worker 的优先级高于 thread1，worker 重新获得了控制，由于它并不需要信号量来完成下面的操作，于是很顺利的它把自己的一大串打印任务都执行完成了，纵然 thread2 的优先级要高于它，但是奈何获取不到信号量，什么也干不了，只能被阻塞而干等，于是实验原理中提到的那一幕便发生了。worker 执行结束后，执行权回到了握有信号量的 thread1 手中，当它完成自己的操作，并且释放信号量后，优先级最高的 thread2 才能继续执行。

这其中所发生的就是优先级反转，低优先级的任务反而抢占了高优先级的任务，这种情况在实时系统中是不允许发生的，下一章会介绍如何解决这类问题；