互斥量基本使用

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

- 了解互斥量的基本使用，初始化，获取，释放互斥量，删除/脱离互斥量
- 熟练使用互斥量相关 API

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

实验设计

本实验的主要设计目的是帮助读者快速了解互斥量相关 API，包括静态互斥量初始化/脱离，动态互斥量创建/删除，互斥量获取/释放相关 API。

源程序说明

本实验对应 kernel_mutex_basic

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互斥量，如果使用静态线程和静态互斥量，则此项不是必要的
- #define RT_USING_MUTEX
  此项必须，开启此项后可以使用互斥量相关 API。
- #define RT_USING_CONSOLE
  此项必须，本实验使用 rt_kprintf 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 中定义一个两个全局变量，分别为静态互斥量数据结构和动态互斥量指针。

全局变量定义代码

```
  /* 互斥量控制块 */
  static struct rt_mutex static_mutex;
  /* 指向互斥量的指针 */
  static rt_mutex_t dynamic_mutex = RT_NULL;
```

在 applications/application.c 中的 int rt_application_init()函数中初始化一个静态互斥量和一个动态互斥量，如下所示

互斥量创建/初始化代码

```
     rt_err_t result;

     /* 初始化静态互斥量 */
     result = rt_mutex_init(&static_mutex, "smutex",
RT_IPC_FLAG_FIFO);
     if (result != RT_EOK)
     {
         rt_kprintf("init static mutex failed.\n");
         return -1;
     }

     /* 创建一个动态互斥量 */
     dynamic_mutex = rt_mutex_create("dmutex", RT_IPC_FLAG_FIFO);
     if (dynamic_mutex == RT_NULL)
     {
         rt_kprintf("create dynamic mutex failed.\n");
         return -1;
  }
```

在 int rt_application_init()创建了两个线程 1，分别成为线程 1 和线程 2，如下所示。读者需要注意，线程 2 的优先级比线程 1 的优先级高。

初始化线程代码

```
     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 5
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,
                "thread2",
                rt_thread_entry2,
                RT_NULL,
                &thread2_stack[0],
                sizeof(thread2_stack),10,5);//线程优先级 10，时间片 5
    rt_thread_startup(&thread2);
```

其线程处理函数如下所示。

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)
{
    rt_err_t result;
    rt_tick_t tick;

/* 1. staic mutex demo */

    /* 试图持有互斥量，最大等待 10 个 OS Tick 后返回 */
    rt_kprintf("thread1 try to get static mutex, wait 10 ticks.\n");

    /* 获得当前的 OS Tick */
    tick = rt_tick_get();
    result = rt_mutex_take(&static_mutex, 10);

    if (result == -RT_ETIMEOUT)
    {
        /* 超时后判断是否刚好是 10 个 OS Tick */
        if (rt_tick_get() - tick != 10)
        {
```

```
            rt_mutex_detach(&static_mutex);
            return;
        }
        rt_kprintf("thread1 take static mutex timeout\n");
    }
    else
    {
        /* 线程 2 持有互斥量，且在相当长的时间后才会释放互斥量，
         * 因此 10 个 tick 后线程 1 不可能获得 */
        rt_kprintf("thread1 take a static mutex, failed.\n");
        rt_mutex_detach(&static_mutex);
        return;
    }

    /* 永久等待方式持有互斥量 */
    rt_kprintf("thread1 try to get static mutex, wait forever.\n");
    result = rt_mutex_take(&static_mutex, RT_WAITING_FOREVER);
    if (result != RT_EOK)
    {
        /* 不成功则测试失败 */
        rt_kprintf("thread1 take a static mutex, failed.\n");
        rt_mutex_detach(&static_mutex);
        return;
    }

    rt_kprintf("thread1 take a staic mutex, done.\n");

    /* 脱离互斥量对象 */
    rt_mutex_detach(&static_mutex);

/* 2. dynamic mutex test */

    /* 试图持有互斥量，最大等待 10 个 OS Tick 后返回 */
    rt_kprintf("thread1 try to get dynamic mutex, wait 10 ticks.\n");

    tick = rt_tick_get();
    result = rt_mutex_take(dynamic_mutex, 10);
    if (result == -RT_ETIMEOUT)
```

```
    {
        /* 超时后判断是否刚好是 10 个 OS Tick */
        if (rt_tick_get() - tick != 10)
        {
            rt_mutex_delete(dynamic_mutex);
            return;
        }
        rt_kprintf("thread1 take dynamic mutex timeout\n");
    }
    else
    {
        /* 线程 2 持有互斥量，且在相当长的时间后才会释放互斥量，
         * 因此 10 个 tick 后线程 1 不可能获得 */
        rt_kprintf("thread1 take a dynamic mutex, failed.\n");
        rt_mutex_delete(dynamic_mutex);
        return;
    }

    /* 永久等待方式持有互斥量 */
    rt_kprintf("thread1 try to get dynamic mutex, wait forever.\n");
    result = rt_mutex_take(dynamic_mutex, RT_WAITING_FOREVER);
    if (result != RT_EOK)
    {
        /* 不成功则测试失败 */
        rt_kprintf("thread1 take a dynamic mutex, failed.\n");
        rt_mutex_delete(dynamic_mutex);
        return;
    }

    rt_kprintf("thread1 take a dynamic mutex, done.\n");
    /* 删除互斥量对象 */
    rt_mutex_delete(dynamic_mutex);
}
```

线程 2 的入口函数如下所示

线程 2 代码

```
ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
```

```
struct rt_thread thread2;
static void rt_thread_entry2(void* parameter)
{
    rt_err_t result;
    rt_tick_t tick;

    /* 1. static mutex test */
    rt_kprintf("thread2 try to get static mutex\n");
    rt_mutex_take(&static_mutex, 10);
    rt_kprintf("thread2 got static mutex\n");
    rt_thread_delay(RT_TICK_PER_SECOND);
    rt_kprintf("thread2 release static mutex\n");
    rt_mutex_release(&static_mutex);

    /* 2. dynamic mutex test */
    rt_kprintf("thread2 try to get dynamic mutex\n");
    rt_mutex_take(dynamic_mutex, 10);
    rt_kprintf("thread2 got dynamic mutex\n");
    rt_thread_delay(RT_TICK_PER_SECOND);
    rt_kprintf("thread2 release dynamic mutex\n");
    rt_mutex_release(dynamic_mutex);
}
```

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：

串口输出信息

```
\ | /
- RT -     Thread Operating System
 / | \     1.1.0 build Aug  9 2012
 2006 - 2012 Copyright by rt-thread team
thread2 try to get static mutex
thread2 got static mutex
thread1 try to get static mutex, wait 10 ticks.
```

```
thread1 take static mutex timeout
thread1 try to get static mutex, wait forever.
thread2 release static mutex
thread2 try to get dynamic mutex
thread2 got dynamic mutex
thread1 take a staic mutex, done.
thread1 try to get dynamic mutex, wait 10 ticks.
thread1 take dynamic mutex timeout
thread1 try to get dynamic mutex, wait forever.
thread2 release dynamic mutex
thread1 take a dynamic mutex, done.
```

结果分析

整个程序运行过程中各个线程的状态变化：

rt_application_init 中创建线程 thread1 和线程 thread2，这两个线程相互获取和释放互斥量来完成信号量的基本测试。由于动态互斥量和静态互斥量的测试过程基本一致，因此这里重点分析静态互斥量的测试过程。

由于 thread2 的优先级高于线程 1 的优先级，因此 thread2 先运行，在其线程处理函数中先获取 static_mutex，并打印信息

```
thread2 try to get static mutex
thread2 got static mutex
```

随后 thread2 休眠 1 秒钟，线程 2 被挂起，线程 1 调度运行。在线程 1 中试图持有 static_mutex，并设定等待时间为 10 个 tick，此时 static_mutex 依然被处于挂起中的线程 2 所持有，因此线程 1 会被挂起。这个过程中打印信息为：

```
thread1 try to get static mutex, wait 10 ticks.
thread1 take static mutex timeout
```

内核调度 IDLE 线程执行，10 个 tick 后，线程 1 依然无法拿到 staic_mutex，等待超时，线程 1 再次被唤醒。接下来线程 1 试图使用

result = rt_mutex_take(&static_mutex, RT_WAITING_FOREVER);

这会永久等待信号量 static_mutex。

打印信息为

```
thread1 try to get static mutex, wait forever.
```

线程 1 会被一直挂起。此时内核调度 IDLE 线程执行

大约 1 秒后，线程 2 被唤醒立刻抢占 IDLE 线程并运行，释放 static_mutex，线程 1 被唤醒，状态变为就绪态，但由于线程 1 优先级低于线程 2，因此线程 2 继续运行，获取动态信号量，这个过程打印如下信息：

```
thread2 release static mutex
thread2 try to get dynamic mutex
thread2 got dynamic mutex
```

接下来线程 2 再次休眠 1 秒钟，之后的过程跟前面的过程类似，就留给读者分析。

总结

本实验演示了 RT-Thread 中互斥量的 API 的基本使用，对于静态互斥量，使用 init/detach 来初始化和脱离，对用动态互斥量使用 create/delete 来创建删除。take/release 中的第一个参数为指针，因此当时使用静态互斥量时，需要取去地址符号&，读者需要注意。