信号量基本使用

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

- 了解信号量的基本使用：初始化，获取信号量，释放信号量，删除/脱离信号量
- 熟练使用信号量相关 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

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信号量相关 API，包括静态信号量初始化/脱离，动态信号量创建/删除，信号量获取释放相关 API，为了简化起见，我们将这些 API 放在同一个线程中调用。请读者注意，本实验本身不具有实际的工程参考价值，只是帮助读者快速了解信号量 API 的用法。

源程序说明

本实验对应 kernel_sem_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 中定义一个两个全局变量，分别为静态信号量数据结构和动态信号量指针。

定义变量

```
/* 信号量控制块 */
static struct rt_semaphore static_sem;
/* 指向信号量的指针 */
static rt_sem_t dynamic_sem = RT_NULL;
```

在 applications/application.c 中的 int rt_application_init()函数中初始化一个静态信号量，创建一个动态信号量，如下所示

创建信号量

```
    /* 初始化静态信号量，初始值是 0 */
    result = rt_sem_init(&static_sem, "ssem", 0, RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
    {
        rt_kprintf("init dynamic semaphore failed.\n");
        return -1;
    }

    /* 创建一个动态信号量，初始值是 0 */
    dynamic_sem = rt_sem_create("dsem", 0, RT_IPC_FLAG_FIFO);
    if (dynamic_sem == RT_NULL)
    {
        rt_kprintf("create dynamic semaphore failed.\n");
        return -1;
    }
```

在 int rt_application_init()名为”thread1”的 thread1 的静态线程，如下所示。

```
    rt_thread_init(&thread1,
                   "thread1",
                   rt_thread_entry1,
```

```
                RT_NULL,
                &thread1_stack[0],
                sizeof(thread1_stack),11,5);//线程优先为11,时间片为5
    rt_thread_startup(&thread1);
```

首先测试静态信号量 API，首先获取静态信号量 static_sem，因为在 rt_application_init 中函数中初始化信号量初值为 0，因此这里 take 会导致线程挂起。由于没有另外的线程释放这个信号量，10 个 tick 之后此线程依然没有无法获取到信号量，rt_sem_take 会超时返回。

静态信号量测试代码片段 1

```
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 semaphore demo */
    /* 获得当前的OS Tick */
    tick = rt_tick_get();

    /* 试图持有信号量，最大等待10个OS Tick后返回 */
    result = rt_sem_take(&static_sem, 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_sem_detach(&static_sem);
            return;
        }
        rt_kprintf("take semaphore timeout\n");
    }
    else
    {
```

```
    /* 因为没有其他地方释放信号量，所以不应该成功持有信号量 */
    rt_kprintf("take a static semaphore, failed.\n");
    rt_sem_detach(&static_sem);
    return;
}
```

继续动态信号量测试，首先释放一次静态信号量，这样信号量的值为 1，然后再次调用 take 函数来试图获取信号量，注意此时的参数为 RT_WAITING_FOREVER，即如果没有取得信号量的话，线程将会一直处于等待挂起状态，直到取得信号量。由于之前已经通过 release 释放了一次信号量，因此这里可以正确取得信号量，之后调用 detach 函数脱离这个静态信号量。

静态信号量测试代码片段 2

```
/* 释放一次信号量 */
rt_sem_release(&static_sem);

/* 永久等待方式持有信号量 */
result = rt_sem_take(&static_sem, RT_WAITING_FOREVER);
if (result != RT_EOK)
{
    /* 不成功则测试失败 */
    rt_kprintf("take a static semaphore, failed.\n");
    rt_sem_detach(&static_sem);
    return;
}

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

/* 脱离信号量对象 */
rt_sem_detach(&static_sem);}
```

接下来测试动态信号量 API，首先获取动态信号量 dynamic_sem，因为在 rt_application_init 中函数中初始化信号量初值为 0，因此这里 take 会导致线程挂起。由于没有另外的线程释放这个信号量，10 个 tick 之后此线程依然没有无法获取到信号量，rt_sem_take 会超时返回。

动态信号量测试代码片段 1

```
    tick = rt_tick_get();

    /* 试图持有信号量，最大等待 10 个 OS Tick 后返回 */
    result = rt_sem_take(dynamic_sem, 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_sem_delete(dynamic_sem);
            return;
        }
        rt_kprintf("take semaphore timeout\n");
    }
    else
    {
        /* 因为没有其他地方释放信号量，所以不应该成功持有信号量，否则测试失败
*/
        rt_kprintf("take a dynamic semaphore, failed.\n");
        rt_sem_delete(dynamic_sem);
        return;
 }
```

继续动态信号量测试，首先释放一次动态信号来量，这样信号量的值为 1，然后再次调用 take 函数来试图获取信号量，注意此时的参数为 RT_WAITING_FOREVER，即如果没有取得信号量的话，线程将会一直处于等待挂起状态，直到取得信号量。由于之前已经通过 release 释放了一次信号量，因此这里可以正确取得信号量，之后调用 delete 函数删除这个动态信号量。

动态信号量测试代码片段 2

```
    /* 释放一次信号量 */
    rt_sem_release(dynamic_sem);

    /* 永久等待方式持有信号量 */
    result = rt_sem_take(dynamic_sem, RT_WAITING_FOREVER);
    if (result != RT_EOK)
```

```
    {
        /* 不成功则测试失败 */
        rt_kprintf("take a dynamic semaphore, failed.\n");
        rt_sem_delete(dynamic_sem);
        return;
    }

    rt_kprintf("take a dynamic semaphore, done.\n");
    /* 删除信号量对象 */
    rt_sem_delete(dynamic_sem);
}
```

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  7 2012
 2006 - 2012 Copyright by rt-thread team
take semaphore timeout
take a staic semaphore, done.
take semaphore timeout
take a dynamic semaphore, 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，在这个线程中完成信号量 API 的使用以及测试，由于动态信号量和静态信号量的测试基本一致，因此这里重点分析静态信号量的测试过程。

thread1 中首先获取静态信号量 static_sem，由于此信号量在 rt_application_init 函数中被初始化为 0，并且没有其他线程做 release

信号量操作，因此在第一句 rt_sem_take 会导致导致 thread1 被挂起，内核调度器会从系统中所有就绪线程中寻找优先级在最高的线程运行，在本实验中只有 IDLE 线程处于就绪态，因此 IDLE 线程运行。10 个 tick 之后，static_sem 超时，rt_sem_take 超时返回，thread1 重新被唤醒变为就绪态，由于其线程优先级高于 IDLE 线程，因此会抢占 IDLE 线程运行。

接下来调用 release 函数释放一个信号量，此时信号量值为 1，再次执行 rt_sem_take 会成功获取信号量。测试完毕后，调用 rt_sem_detach 脱离静态信号量。

总结

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