使用信号量解决生产者消费者问题

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

- 了解什么是生产者消费者问题
- 学习信号量的互斥功能
- 学习信号量的同步功能
- 学习使用信号量来解决生产者消费者问题

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

生产者消费者问题是一个著名的线程同步问题，该问题描述如下：有一个生产者在生产产品，这些产品将提供给若干个消费者去消费。为了使生产者和消费者能并发执行，在两者之间设置一个具有多个位置的缓冲区，生产者将它生产的产品放入缓冲区中，消费者可以从缓冲区中取走产品进行消费。生产者和消费者之间必须保持同步，即当缓冲区为空时，消费者需要被阻塞（即挂起）直到生产者者生产出产品并放入缓冲区；当缓冲区满时，生产者则需要被阻塞直到消费者冲缓冲区取走产品使得缓冲区有至少一个空位。

我们以打印机例子来说明这个问题。一个打印系统中，一个任务负责向打印系统的打印队列中加入需要的打印的文件，而打印系统中有两台打印机，它们可以同时从打印队列中获取文件并启动打印。

在设计程序时，读者需要注意：

- 从缓冲区取出产品和向缓冲区投放产品的过程为临界区，必须是互斥进行的，可以使用互斥量（mutex）、二值信号量（即信

号量的值只为 0 或 1）或调度器上锁实现临界区互斥，本例中使用二值信号量。

- 当缓冲区不满时，生产者才可以向缓冲区中投放产品；当缓冲区不空时，消费者才可以从缓冲区中取出产品消费，因此这意味着生产者和消费者线程各自获取一个信号量。

在 RT-Thread 中，我们创建一个生产者线程，另外创建两个线程则分别以一定周期（10 个 tick）来从缓冲区中取出产品消费。

源程序说明

本实验对应 kernel_sem_producer_consumer

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_SEMAPHORE
  此项必需，开启后才可以使用信号量
- #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 中定义一些全局变量，如下所示

定义全局变量代码

```
/* 定义最大 5 个元素能够被产生 */
#define MAXSEM 5
/* 用于放置生产的整数数组 */
rt_uint32_t array[MAXSEM];
/* 指向生产者、消费者在 array 数组中的读写位置 */
static rt_uint32_t set, get;
/* 定义二值信号量实现临界区互斥 */
struct rt_semaphore sem_lock;
/* 定义信号量用于生产者和消费者线程 */
struct rt_semaphore sem_empty, sem_full;
```

在 applications/application.c 中的 int rt_application_init()函数中创建三个信号量，其中，sem_lock 为二值型信号量，其初值为 1，在本实验中它用来实现临界区互斥。

sem_empty 信号量用来表示缓冲区中空闲的位置数目，开始时，缓冲区为空，因此初始值为缓冲区中位置数目。

sem_full 信号量用来表示缓冲区中非空闲的位置数目，开始时缓冲区为空，因此初值为 0。

初始化信号量代码

```
    rt_err_t result;
    /* 初始化 3 个信号量 */
    result = rt_sem_init(&sem_lock , "lock", 1, RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
        goto _error;
    result = rt_sem_init(&sem_empty, "empty", MAXSEM,
RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
        goto _error;
    result = rt_sem_init(&sem_full , "full", 0, RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
        goto _error;
```

接下来创建并启动三个线程。

创建线程代码

```
    tid = rt_thread_create(
                "producer",
                producer_thread_entry,
                RT_NULL,
                THREAD_STACK_SIZE, 10, 5);//生产者优先级别为 10
    if(tid != RT_NULL)
        rt_thread_startup(tid);

    tid = rt_thread_create(
                "consumer1",
                consumer_thread_entry,
                (void *)1, //消费者线程 1
```

```
                THREAD_STACK_SIZE, 11, 2);//消费者线程优先级为11
    if(tid != RT_NULL)
        rt_thread_startup(tid);

    tid = rt_thread_create(
                "consumer2",
                consumer_thread_entry,
                (void *)2,//消费者线程2
                THREAD_STACK_SIZE, 11, 2);//消费者线程优先级为11
    if(tid != RT_NULL)
        rt_thread_startup(tid);

return 0;
_error:
    rt_kprintf("init semaphore failed.\n");
    return -1;
}
```

生产者线程处理函数如下所示，生产者线程需要先获取 sem_empty，当成功获取，这表示缓冲区中有空位，即可以向缓冲区中投放“产品”。注意修改缓冲区的代码为临界区，因此使用二值信号量 sem_lock 将整个临界区保护起来。

生产者线程函数代码

```
/* 生产者线程入口 */
void producer_thread_entry(void* parameter)
{
    int cnt = 0;

    /* 运行20次 */
    while (cnt < 20)
    {
        /* 获取一个空位 */
        rt_sem_take(&sem_empty, RT_WAITING_FOREVER);

        /* 修改array内容，上锁 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        array[set%MAXSEM] = cnt + 1;
```

```
        rt_kprintf("the producer generates a number: %d\n",
array[set%MAXSEM]);
        set++;
        rt_sem_release(&sem_lock);

        /* 发布一个满位 */
        rt_sem_release(&sem_full);
        cnt++;

        /* 暂停一段时间 */
        //rt_thread_delay(15);
    }

    rt_kprintf("the producer exit!\n");
}
```

消费者线程如下所示，为了减少代码的体积，这两个消费者线程采了相同的线程入口函数。这两个线程通过参数 parameter 来区分开来，在线程创建时指定分别制定了消费者线程 1 和消费者线程 2。

消费者线程中，首先需要获取 sem_full 信号量，如果其值为 0 则意味着缓冲区中没有产品，rt_sem_take 会导致线程阻塞直到生产者向缓冲区投入产品（即发布 sem_full 信号量）。如果成功获取 sem_full 信号量，这表示缓冲区非空，即可以取出产品消费。之后，执行 rt_sem_release(sem_emptry)，缓冲区的空位数目增加一个。同样，在对缓冲区操作的过程为临界区操作，同样需要使用 sem_lock 进行互斥保护。

消费者线程函数代码

```
/* 消费者线程入口 */
void consumer_thread_entry(void* parameter)
{
    rt_uint32_t no;
    rt_uint32_t sum;

    /* 第 n 个线程，由入口参数传进来 */
    no = (rt_uint32_t)parameter;
```

```
    sum = 0;

    while(1)
    {
        /* 获取一个满位 */
        rt_sem_take(&sem_full, RT_WAITING_FOREVER);

        /* 临界区，上锁进行操作 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        sum += array[get%MAXSEM];
        rt_kprintf("the consumer[%d] get a number: %d\n", no,
array[get%MAXSEM] );
        get++;
        rt_sem_release(&sem_lock);

        /* 释放一个空位 */
        rt_sem_release(&sem_empty);

        /* 生产者生产到 20 个数目，停止，消费者线程相应停止 */
        if (get == 20)
            break;

        /* 暂停一小会时间 */
        rt_thread_delay(10);
    }

    rt_kprintf("the consumer[%d] exits, sum is %d \n ", no, sum);
 }
```

编译调试及观察输出信息

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
the producer generates a number: 1
the producer generates a number: 2
the producer generates a number: 3
the producer generates a number: 4
the producer generates a number: 5
the consumer[1] get a number: 1
the consumer[2] get a number: 2
the producer generates a number: 6
the consumer[1] get a number: 3
the producer generates a number: 7
the consumer[2] get a number: 4
the producer generates a number: 8
the consumer[1] get a number: 5
the producer generates a number: 9
the consumer[2] get a number: 6
the producer generates a number: 10
the consumer[1] get a number: 7
the producer generates a number: 11
the consumer[2] get a number: 8
the producer generates a number: 12
the consumer[1] get a number: 9
the producer generates a number: 13
the consumer[2] get a number: 10
the producer generates a number: 14
the consumer[1] get a number: 11
the producer generates a number: 15
the consumer[2] get a number: 12
the producer generates a number: 16
the consumer[1] get a number: 13
the producer generates a number: 17
the consumer[2] get a number: 14
the producer generates a number: 18
the consumer[1] get a number: 15
the producer generates a number: 19
the consumer[2] get a number: 16
the producer generates a number: 20
```

```
the producer exit!
the consumer[1] get a number: 17
the consumer[2] get a number: 18
the consumer[1] get a number: 19
the consumer[2] get a number: 20
```

结果分析

读者需要注意生产者线程的入口函数没有 rt_thread_delay，而消费者线程入口函数中各自 delay 了 10 个 tick。并且生产者线程的优先级比消费者优先级的线程稍高，这就意味着生产者线程的产生的速度要快于两个消费者线程消费的速度。因此在串口的开始的打印信息中，

```
the producer generates a number: 1
the producer generates a number: 2
the producer generates a number: 3
the producer generates a number: 4
the producer generates a number: 5
```

消费者会将整个缓冲区填满，直到缓冲区没有空位，生产者线程被挂起，消费者线程启动。

接下来的打印数据如下，这个信息是很需要注意的。

```
the consumer[1] get a number: 1
the consumer[2] get a number: 2
```

消费者线程 1 启动后从缓冲区中取出一个“产品”发送至串口，这个过程是临界区，由二值信号量 sem_lock 保护。消费者线程 1 和消费者线程 2 的优先级是相同的，意味着它们是按照时间片轮换运行的。

那么问题来了，消费者 1 的在临界区运行的时间是否会超过其线程时间片呢？

从上面的打印信息来看应该是超过了，因为如果消费者 1 执行完毕后继续执行 rt_sem_release(&sem_empty)发布信号量，这会导致生产者线程立刻被唤醒并抢占消费者线程运行，即消费者线程 2 是根本没有机会运行的，但是上面的打印的信息中，消费者 2 线程也打印了，这说明消费者 1

线程在临界区执行的过程中时间片就已经耗尽，此时消费者线程 2 调度运行，但是它在临界区处执行 rt_sem_take(sem_lock)时，此时 sem_lock 已经被消费者线程 1 获取，sem_lock 的值为 0，因此消费者线程 2 被挂起。内核再次切换至消费者线程 1 的线程入口函数继续执行，当执行 rt_sem_release(&sem_lock)退出临界区后，消费者 2 线程立刻被唤醒进入临界区执行，并打印 the consumer[2] get a number: 2。

之后的情况更加复杂，就不再进一步分析了。

总结

本实验演示了信号量的两种用法，当信号量值为 1 时，该信号量类似于 mutex，具有互斥功能。当信号量的值大于 1 时，则具有计数功能，并可以实现多个线程之间的同步。

本实验详细解释了生产者消费者问题，并给出了使用信号量的解决方案。再次强调，如果使用互斥量 mutex 来保护临界区会比本实验中使用的互斥信号量 sem_lock 更好，程序上更加的清晰。