线程间同步

RT-THREAD 文档中心

WWW.RT-THREAD.ORG

Tuesday 25th July, 2023

目录

在多线程实时系统中，一项工作的完成往往可以通过多个线程协调的方式共同来完成，那么多个线程之间如何 “默契” 协作才能使这项工作无差错执行？下面举个例子说明。

例如一项工作中的两个线程：一个线程从传感器中接收数据并且将数据写到共享内存中，同时另一个线程周期性的从共享内存中读取数据并发送去显示，下图描述了两个线程间的数据传递：

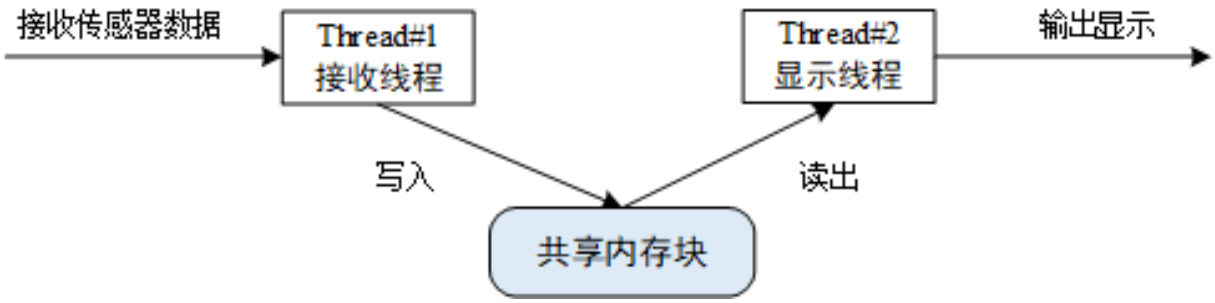

图 1: *线程间数据传递示意图*

如果对共享内存的访问不是排他性的，那么各个线程间可能同时访问它，这将引起数据一致性的问题。例如，在显示线程试图显示数据之前，接收线程还未完成数据的写入，那么显示将包含不同时间采样的数据，造成显示数据的错乱。

将传感器数据写入到共享内存块的接收线程 #1 和将传感器数据从共享内存块中读出的线程 #2 都会访问同一块内存。为了防止出现数据的差错，两个线程访问的动作必须是互斥进行的，应该是在一个线程对共享内存块操作完成后，才允许另一个线程去操作，这样，接收线程 #1 与显示线程 #2 才能正常配合，使此项工作正确地执行。

同步是指按预定的先后次序进行运行，线程同步是指多个线程通过特定的机制（如互斥量，事件对象，临界区）来控制线程之间的执行顺序，也可以说是在线程之间通过同步建立起执行顺序的关系，如果没有同步，那线程之间将是无序的。

多个线程操作 / 访问同一块区域（代码），这块代码就称为临界区，上述例子中的共享内存块就是临界区。线程互斥是指对于临界区资源访问的排它性。当多个线程都要使用临界区资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

线程的同步方式有很多种，其核心思想都是：**在访问临界区的时候只允许一个 (或一类) 线程运行**。进入 / 退出临界区的方式有很多种：

1）调用 rt_hw_interrupt_disable() 进入临界区，调用 rt_hw_interrupt_enable() 退出临界区；详见《中断管理》的全局中断开关内容。

2）调用 rt_enter_critical() 进入临界区，调用 rt_exit_critical() 退出临界区。

本章将介绍多种同步方式：**信号量**（semaphore）、**互斥量**（mutex）、和**事件集**（event）。学习完本章，大家将学会如何使用信号量、互斥量、事件集这些对象进行线程间的同步。

1 信号量

以生活中的停车场为例来理解信号量的概念：

 当停车场空的时候，停车场的管理员发现有很多空车位，此时会让外面的车陆续进入停车场获得停车位；

 当停车场的车位满的时候，管理员发现已经没有空车位，将禁止外面的车进入停车场，车辆在外排队等候；

 当停车场内有车离开时，管理员发现有空的车位让出，允许外面的车进入停车场；待空车位填满后，又禁止外部车辆进入。

在此例子中，管理员就相当于信号量，管理员手中空车位的个数就是信号量的值（非负数，动态变化）；停车位相当于公共资源（临界区），车辆相当于线程。车辆通过获得管理员的允许取得停车位，就类似于线程通过获得信号量访问公共资源。

1.1 信号量工作机制

信号量是一种轻型的用于解决线程间同步问题的内核对象，线程可以获取或释放它，从而达到同步或互斥的目的。

信号量工作示意图如下图所示，每个信号量对象都有一个信号量值和一个线程等待队列，信号量的值对应了信号量对象的实例数目、资源数目，假如信号量值为 5，则表示共有 5 个信号量实例（资源）可以被使用，当信号量实例数目为零时，再申请该信号量的线程就会被挂起在该信号量的等待队列上，等待可用的信号量实例（资源）。

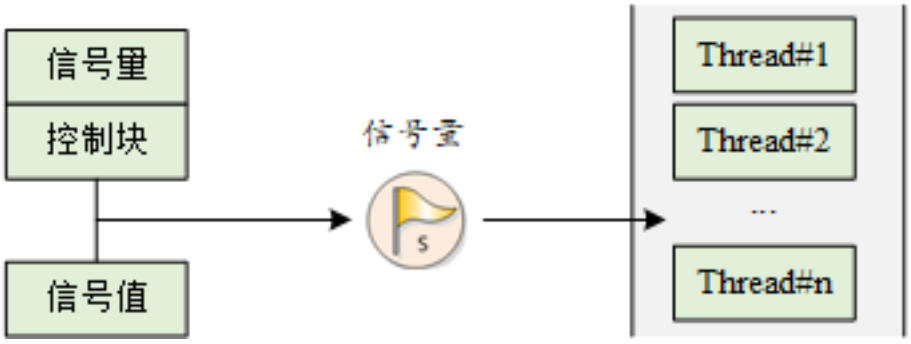

图 2: *信号量工作示意图*

1.2 信号量控制块

在 RT-Thread 中，信号量控制块是操作系统用于管理信号量的一个数据结构，由结构体 struct rt_semaphore 表示。另外一种 C 表达方式 rt_sem_t，表示的是信号量的句柄，在 C 语言中的实现是指向信号量控制块的指针。信号量控制块结构的详细定义如下：

```
struct rt_semaphore
{
    struct rt_ipc_object parent;  /* 继承自 ipc_object 类 */
    rt_uint16_t value;            /* 信号量的值 */
};
/* rt_sem_t 是指向 semaphore 结构体的指针类型 */
typedef struct rt_semaphore* rt_sem_t;
```

rt_semaphore 对象从 rt_ipc_object 中派生，由 IPC 容器所管理，信号量的最大值是 65535。

1.3 信号量的管理方式

信号量控制块中含有信号量相关的重要参数，在信号量各种状态间起到纽带的作用。信号量相关接口如下图所示，对一个信号量的操作包含：创建 / 初始化信号量、获取信号量、释放信号量、删除 / 脱离信号量。

图 3: *信号量相关接口*

1.3.1. 创建和删除信号量

当创建一个信号量时，内核首先创建一个信号量控制块，然后对该控制块进行基本的初始化工作，创建信号量使用下面的函数接口：

```
rt_sem_t rt_sem_create(const char *name,
                       rt_uint32_t value,
                       rt_uint8_t flag);
```

当调用这个函数时，系统将先从对象管理器中分配一个 semaphore 对象，并初始化这个对象，然后初始化父类 IPC 对象以及与 semaphore 相关的部分。在创建信号量指定的参数中，信号量标志参数决定了当信号量不可用时，多个线程等待的排队方式。当选择 RT_IPC_FLAG_FIFO（先进先出）方式时，那么等待线程队列将按照先进先出的方式排队，先进入的线程将先获得等待的信号量；当选择 RT_IPC_FLAG_PRIO（优先级等待）方式时，等待线程队列将按照优先级进行排队，优先级高的等待线程将先获得等待的信号量。

> 注：RT_IPC_FLAG_FIFO 属于非实时调度方式，除非应用程序非常在意先来后到，并且你清楚地明白所有涉及到该信号量的线程都将会变为非实时线程，方可使用 RT_IPC_FLAG_FIFO，否则建议采用 RT_IPC_FLAG_PRIO，即确保线程的实时性。

下表描述了该函数的输入参数与返回值：

参数	描述
name	信号量名称
value	信号量初始值

参数	描述
flag	信号量标志，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	——
RT_NULL	创建失败
信号量的控制块指针	创建成功

系统不再使用信号量时，可通过删除信号量以释放系统资源，适用于动态创建的信号量。删除信号量使用下面的函数接口：

```
rt_err_t rt_sem_delete(rt_sem_t sem);
```

调用这个函数时，系统将删除这个信号量。如果删除该信号量时，有线程正在等待该信号量，那么删除操作会先唤醒等待在该信号量上的线程（等待线程的返回值是 - RT_ERROR），然后再释放信号量的内存资源。下表描述了该函数的输入参数与返回值：

rt_sem_delete() 的输入参数和返回值

参数	描述
sem	rt_sem_create() 创建的信号量对象
返回	——
RT_EOK	删除成功

1.3.2. 初始化和脱离信号量

对于静态信号量对象，它的内存空间在编译时期就被编译器分配出来，放在读写数据段或未初始化数据段上，此时使用信号量就不再需要使用 rt_sem_create 接口来创建它，而只需在使用前对它进行初始化即可。初始化信号量对象可使用下面的函数接口：

```
rt_err_t rt_sem_init(rt_sem_t       sem,
                     const char     *name,
                     rt_uint32_t    value,
                     rt_uint8_t     flag)
```

当调用这个函数时，系统将对这个 semaphore 对象进行初始化，然后初始化 IPC 对象以及与 semaphore 相关的部分。信号量标志可用上面创建信号量函数里提到的标志。下表描述了该函数的输入参数与返回值：

rt_sem_init() 的输入参数和返回值

参数	描述
sem	信号量对象的句柄

参数	描述
name	信号量名称
value	信号量初始值
flag	信号量标志，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	——
RT_EOK	初始化成功

脱离信号量就是让信号量对象从内核对象管理器中脱离，适用于静态初始化的信号量。脱离信号量使用下面的函数接口：

```
rt_err_t rt_sem_detach(rt_sem_t sem);
```

使用该函数后，内核先唤醒所有挂在该信号量等待队列上的线程，然后将该信号量从内核对象管理器中脱离。原来挂起在信号量上的等待线程将获得 - RT_ERROR 的返回值。下表描述了该函数的输入参数与返回值：

rt_sem_detach() 的输入参数和返回值

参数	描述
sem	信号量对象的句柄
返回	——
RT_EOK	脱离成功

1.3.3. 获取信号量

线程通过获取信号量来获得信号量资源实例，当信号量值大于零时，线程将获得信号量，并且相应的信号量值会减 1，获取信号量使用下面的函数接口：

```
rt_err_t rt_sem_take (rt_sem_t sem, rt_int32_t time);
```

在调用这个函数时，如果信号量的值等于零，那么说明当前信号量资源实例不可用，申请该信号量的线程将根据 time 参数的情况选择直接返回、或挂起等待一段时间、或永久等待，直到其他线程或中断释放该信号量。如果在参数 time 指定的时间内依然得不到信号量，线程将超时返回，返回值是 - RT_ETIMEOUT。下表描述了该函数的输入参数与返回值：

rt_sem_take() 的输入参数和返回值

参数	描述
sem	信号量对象的句柄
time	指定的等待时间，单位是操作系统时钟节拍（OS Tick）
返回	——

参数	描述
RT_EOK	成功获得信号量
-RT_ETIMEOUT	超时依然未获得信号量
-RT_ERROR	其他错误

1.3.4. 无等待获取信号量

当用户不想在申请的信号量上挂起线程进行等待时，可以使用无等待方式获取信号量，无等待获取信号量使用下面的函数接口：

```
rt_err_t rt_sem_trytake(rt_sem_t sem);
```

这个函数与 rt_sem_take(sem, 0) 的作用相同，即当线程申请的信号量资源实例不可用的时候，它不会等待在该信号量上，而是直接返回 - RT_ETIMEOUT。下表描述了该函数的输入参数与返回值：

rt_sem_trytake() 的输入参数和返回值

参数	描述
sem	信号量对象的句柄
返回	——
RT_EOK	成功获得信号量
-RT_ETIMEOUT	获取失败

1.3.5. 释放信号量

释放信号量可以唤醒挂起在该信号量上的线程。释放信号量使用下面的函数接口：

```
rt_err_t rt_sem_release(rt_sem_t sem);
```

例如当信号量的值等于零时，并且有线程等待这个信号量时，释放信号量将唤醒等待在该信号量线程队列中的第一个线程，由它获取信号量；否则将把信号量的值加 1。下表描述了该函数的输入参数与返回值：

rt_sem_release() 的输入参数和返回值

参数	描述
sem	信号量对象的句柄
返回	——
RT_EOK	成功释放信号量

1.4 信号量应用示例

这是一个信号量使用例程，该例程创建了一个动态信号量，初始化两个线程，一个线程发送信号量，一个线程接收到信号量后，执行相应的操作。如下代码所示：

信号量的使用

```
#include <rtthread.h>

#define THREAD_PRIORITY         25
#define THREAD_TIMESLICE        5

/* 指向信号量的指针 */
static rt_sem_t dynamic_sem = RT_NULL;

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;
static void rt_thread1_entry(void *parameter)
{
    static rt_uint8_t count = 0;

    while(1)
    {
        if(count <= 100)
        {
            count++;
        }
        else
            return;

        /* count 每计数 10 次，就释放一次信号量 */
         if(0 == (count % 10))
        {
            rt_kprintf("t1 release a dynamic semaphore.\n");
            rt_sem_release(dynamic_sem);
        }
    }
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
static void rt_thread2_entry(void *parameter)
{
    static rt_err_t result;
    static rt_uint8_t number = 0;
    while(1)
    {
        /* 永久方式等待信号量，获取到信号量，则执行 number 自加的操作 */
```

```
        result = rt_sem_take(dynamic_sem, RT_WAITING_FOREVER);
        if (result != RT_EOK)
        {
            rt_kprintf("t2 take a dynamic semaphore, failed.\n");
            rt_sem_delete(dynamic_sem);
            return;
        }
        else
        {
            number++;
            rt_kprintf("t2 take a dynamic semaphore. number = %d\n" ,number);
        }
    }
}

/* 信号量示例的初始化 */
int semaphore_sample(void)
{
    /* 创建一个动态信号量，初始值是 0 */
    dynamic_sem = rt_sem_create("dsem", 0, RT_IPC_FLAG_PRIO);
    if (dynamic_sem == RT_NULL)
    {
        rt_kprintf("create dynamic semaphore failed.\n");
        return -1;
    }
    else
    {
        rt_kprintf("create done. dynamic semaphore value = 0.\n");
    }

    rt_thread_init(&thread1,
                   "thread1",
                   rt_thread1_entry,
                   RT_NULL,
                   &thread1_stack[0],
                   sizeof(thread1_stack),
                   THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,
                   "thread2",
                   rt_thread2_entry,
                   RT_NULL,
                   &thread2_stack[0],
                   sizeof(thread2_stack),
                   THREAD_PRIORITY-1, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);

    return 0;
```

```
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(semaphore_sample, semaphore sample);
```

仿真运行结果：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Aug 27 2018
 2006 - 2018 Copyright by rt-thread team
msh >semaphore_sample
create done. dynamic semaphore value = 0.
msh >t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 1
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 2
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 3
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 4
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 5
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 6
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 7
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 8
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 9
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 10
```

如上面运行结果：线程 1 在 count 计数为 10 的倍数时（count 计数为 100 之后线程退出），发送一个信号量，线程 2 在接收信号量后，对 number 进行加 1 操作。

信号量的另一个应用例程如下所示，本例程将使用 2 个线程、3 个信号量实现生产者与消费者的例子。其中：

3 个信号量分别为：lock：信号量锁的作用，因为 2 个线程都会对同一个数组 array 进行操作，所以该数组是一个共享资源，锁用来保护这个共享资源。empty：空位个数，初始化为 5 个空位。full：满位个数，初始化为 0 个满位。

2 个线程分别为： 生产者线程：获取到空位后，产生一个数字，循环放入数组中，然后释放一个满位。 消费者线程：获取到满位后，读取数组内容并相加，然后释放一个空位。

生产者消费者例程

```
#include <rtthread.h>

#define THREAD_PRIORITY      6
```

```
#define THREAD_STACK_SIZE     512
#define THREAD_TIMESLICE      5

/* 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;

/* 指向线程控制块的指针 */
static rt_thread_t producer_tid = RT_NULL;
static rt_thread_t consumer_tid = RT_NULL;

struct rt_semaphore sem_lock;
struct rt_semaphore sem_empty, sem_full;

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

    /* 运行 10 次 */
    while (cnt < 10)
    {
        /* 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_mdelay(20);
    }

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

/* 消费者线程入口 */
```

```
void consumer_thread_entry(void *parameter)
{
    rt_uint32_t 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", (get % MAXSEM), 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 10 个数目，停止，消费者线程相应停止 */
        if (get == 10) break;

        /* 暂停一小会时间 */
        rt_thread_mdelay(50);
    }

    rt_kprintf("the consumer sum is: %d\n", sum);
    rt_kprintf("the consumer exit!\n");
}

int producer_consumer(void)
{
    set = 0;
    get = 0;

    /* 初始化 3 个信号量 */
    rt_sem_init(&sem_lock, "lock",     1,      RT_IPC_FLAG_PRIO);
    rt_sem_init(&sem_empty, "empty",   MAXSEM, RT_IPC_FLAG_PRIO);
    rt_sem_init(&sem_full, "full",     0,      RT_IPC_FLAG_PRIO);

    /* 创建生产者线程 */
    producer_tid = rt_thread_create("producer",
                                    producer_thread_entry, RT_NULL,
                                    THREAD_STACK_SIZE,
                                    THREAD_PRIORITY - 1,
                                    THREAD_TIMESLICE);
    if (producer_tid != RT_NULL)
    {
```

```
        rt_thread_startup(producer_tid);
    }
    else
    {
        rt_kprintf("create thread producer failed");
        return -1;
    }

    /* 创建消费者线程 */
    consumer_tid = rt_thread_create("consumer",
                                    consumer_thread_entry, RT_NULL,
                                    THREAD_STACK_SIZE,
                                    THREAD_PRIORITY + 1,
                                    THREAD_TIMESLICE);
    if (consumer_tid != RT_NULL)
    {
        rt_thread_startup(consumer_tid);
    }
    else
    {
        rt_kprintf("create thread consumer failed");
        return -1;
    }

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(producer_consumer, producer_consumer sample);
```

该例程的仿真结果如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Aug 27 2018
 2006 - 2018 Copyright by rt-thread team
msh >producer_consumer
the producer generates a number: 1
the consumer[0] get a number: 1
msh >the producer generates a number: 2
the producer generates a number: 3
the consumer[1] get a number: 2
the producer generates a number: 4
the producer generates a number: 5
the producer generates a number: 6
the consumer[2] get a number: 3
the producer generates a number: 7
the producer generates a number: 8
the consumer[3] get a number: 4
```

```
the producer generates a number: 9
the consumer[4] get a number: 5
the producer generates a number: 10
the producer exit!
the consumer[0] get a number: 6
the consumer[1] get a number: 7
the consumer[2] get a number: 8
the consumer[3] get a number: 9
the consumer[4] get a number: 10
the consumer sum is: 55
the consumer exit!
```

本例程可以理解为生产者生产产品放入仓库，消费者从仓库中取走产品。

（1）生产者线程：

1）获取 1 个空位（放产品 number），此时空位减 1；

2）上锁保护；本次的产生的 number 值为 cnt+1，把值循环存入数组 array 中；再开锁；

3）释放 1 个满位（给仓库中放置一个产品，仓库就多一个满位），满位加 1；

（2）消费者线程：

1）获取 1 个满位（取产品 number），此时满位减 1；

2）上锁保护；将本次生产者生产的 number 值从 array 中读出来，并与上次的 number 值相加；再开锁；

3）释放 1 个空位（从仓库上取走一个产品，仓库就多一个空位），空位加 1。

生产者依次产生 10 个 number，消费者依次取走，并将 10 个 number 的值求和。信号量锁 lock 保护 array 临界区资源：保证了消费者每次取 number 值的排他性，实现了线程间同步。

1.5 信号量的使用场合

信号量是一种非常灵活的同步方式，可以运用在多种场合中。形成锁、同步、资源计数等关系，也能方便的用于线程与线程、中断与线程间的同步中。

1.5.1. 线程同步

线程同步是信号量最简单的一类应用。例如，使用信号量进行两个线程之间的同步，信号量的值初始化成 0，表示具备 0 个信号量资源实例；而尝试获得该信号量的线程，将直接在这个信号量上进行等待。

当持有信号量的线程完成它处理的工作时，释放这个信号量，可以把等待在这个信号量上的线程唤醒，让它执行下一部分工作。这类场合也可以看成把信号量用于工作完成标志：持有信号量的线程完成它自己的工作，然后通知等待该信号量的线程继续下一部分工作。

1.5.2. 锁

锁，单一的锁常应用于多个线程间对同一共享资源（即临界区）的访问。信号量在作为锁来使用时，通常应将信号量资源实例初始化成 1，代表系统默认有一个资源可用，因为信号量的值始终在 1 和 0 之间变

动，所以这类锁也叫做二值信号量。如下图所示，当线程需要访问共享资源时，它需要先获得这个资源锁。当这个线程成功获得资源锁时，其他打算访问共享资源的线程会由于获取不到资源而挂起，这是因为其他线程在试图获取这个锁时，这个锁已经被锁上（信号量值是 0）。当获得信号量的线程处理完毕，退出临界区时，它将会释放信号量并把锁解开，而挂起在锁上的第一个等待线程将被唤醒从而获得临界区的访问权。

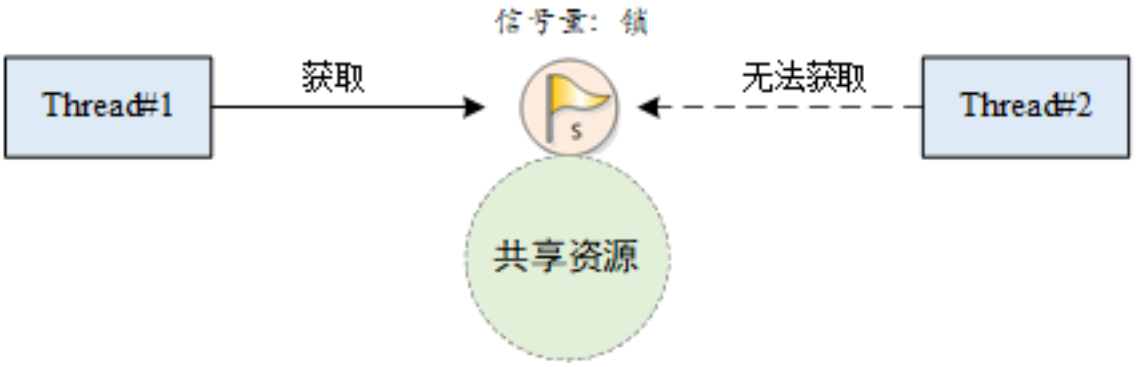

图 4: *锁*

1.5.3. 中断与线程的同步

信号量也能够方便地应用于中断与线程间的同步，例如一个中断触发，中断服务例程需要通知线程进行相应的数据处理。这个时候可以设置信号量的初始值是 0，线程在试图持有这个信号量时，由于信号量的初始值是 0，线程直接在这个信号量上挂起直到信号量被释放。当中断触发时，先进行与硬件相关的动作，例如从硬件的 I/O 口中读取相应的数据，并确认中断以清除中断源，而后释放一个信号量来唤醒相应的线程以做后续的数据处理。例如 FinSH 线程的处理方式，如下图所示。

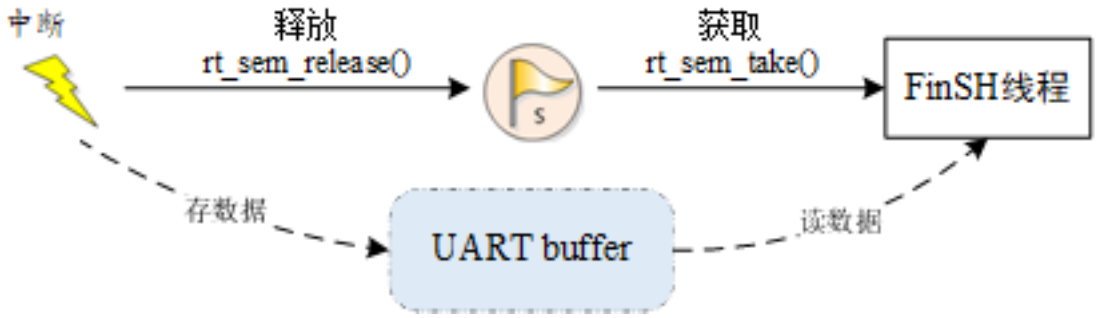

图 5: *FinSH 的中断、线程间同步示意图*

信号量的值初始为 0，当 FinSH 线程试图取得信号量时，因为信号量值是 0，所以它会被挂起。当 console 设备有数据输入时，产生中断，从而进入中断服务例程。在中断服务例程中，它会读取 console 设备的数据，并把读得的数据放入 UART buffer 中进行缓冲，而后释放信号量，释放信号量的操作将唤醒 shell 线程。在中断服务例程运行完毕后，如果系统中没有比 shell 线程优先级更高的就绪线程存在时，shell 线程将持有信号量并运行，从 UART buffer 缓冲区中获取输入的数据。

注意：中断与线程间的互斥不能采用信号量（锁）的方式，而应采用开关中断的方式。

1.5.4. 资源计数

信号量也可以认为是一个递增或递减的计数器，需要注意的是信号量的值非负。例如：初始化一个信号量的值为 5，则这个信号量可最大连续减少 5 次，直到计数器减为 0。资源计数适合于线程间工作处理速

度不匹配的场合，这个时候信号量可以做为前一线程工作完成个数的计数，而当调度到后一线程时，它也可以以一种连续的方式一次处理多个事件。例如，生产者与消费者问题中，生产者可以对信号量进行多次释放，而后消费者被调度到时能够一次处理多个信号量资源。

> 注意：一般资源计数类型多是混合方式的线程间同步，因为对于单个的资源处理依然存在线程的多重访问，这就需要对一个单独的资源进行访问、处理，并进行锁方式的互斥操作。

2 互斥量

互斥量又叫相互排斥的信号量，是一种特殊的二值信号量。互斥量类似于只有一个车位的停车场：当有一辆车进入的时候，将停车场大门锁住，其他车辆在外面等候。当里面的车出来时，将停车场大门打开，下一辆车才可以进入。

2.1 互斥量工作机制

互斥量和信号量不同的是：拥有互斥量的线程拥有互斥量的所有权，互斥量支持递归访问且能防止线程优先级翻转；并且互斥量只能由持有线程释放，而信号量则可以由任何线程释放。

互斥量的状态只有两种，开锁或闭锁（两种状态值）。当有线程持有它时，互斥量处于闭锁状态，由这个线程获得它的所有权。相反，当这个线程释放它时，将对互斥量进行开锁，失去它的所有权。当一个线程持有互斥量时，其他线程将不能够对它进行开锁或持有它，持有该互斥量的线程也能够再次获得这个锁而不被挂起，如下图时所示。这个特性与一般的二值信号量有很大的不同：在信号量中，因为已经不存在实例，线程递归持有会发生主动挂起（最终形成死锁）。

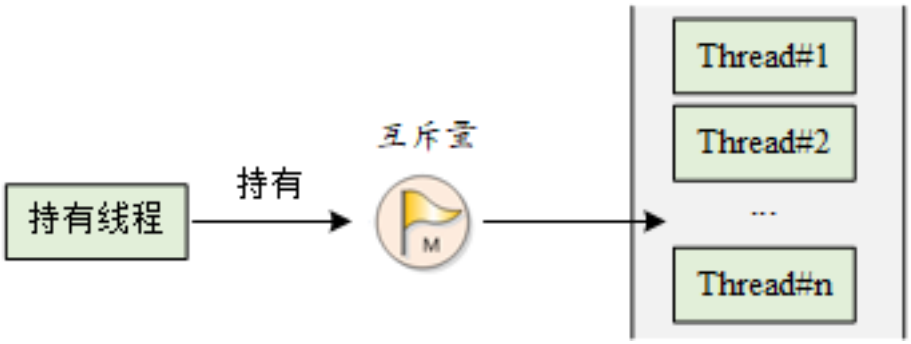

图 6: *互斥量工作示意图*

使用信号量会导致的另一个潜在问题是线程优先级翻转问题。所谓优先级翻转，即当一个高优先级线程试图通过信号量机制访问共享资源时，如果该信号量已被一低优先级线程持有，而这个低优先级线程在运行过程中可能又被其它一些中等优先级的线程抢占，因此造成高优先级线程被许多具有较低优先级的线程阻塞，实时性难以得到保证。如下图所示：有优先级为 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，等待某一事件触发，线程 C 正在运行，此时线程 C 开始使用某一共享资源 M。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 M 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 M 后，线程 A 才得以执行。在这种情况下，优先级发生了翻转：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。

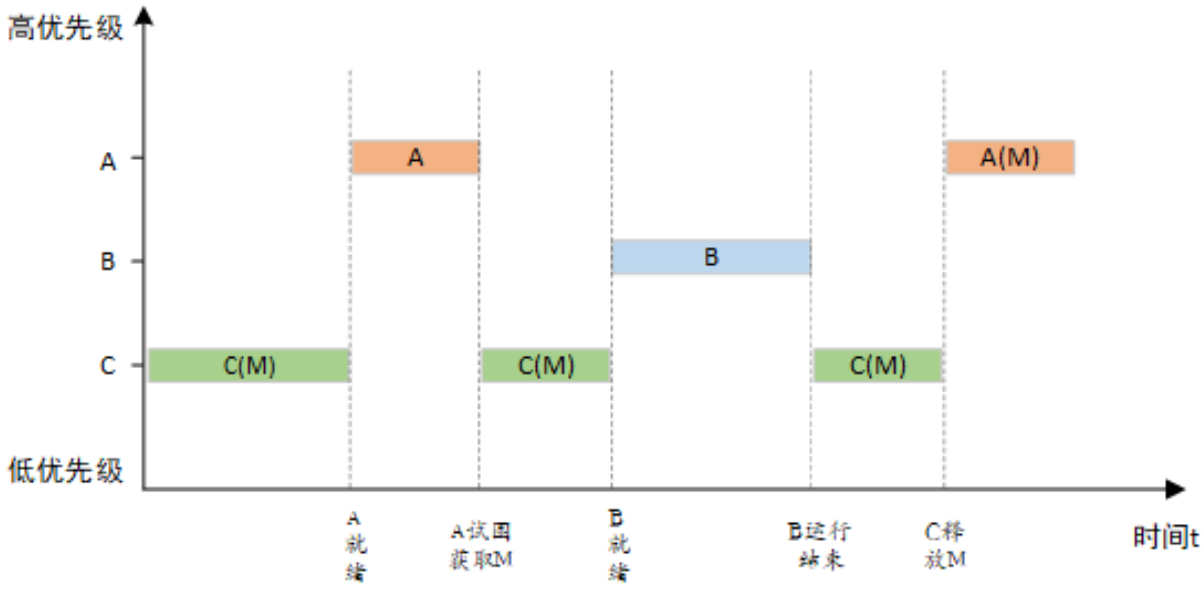

图 7: *优先级反转 (M 为信号量)*

在 RT-Thread 操作系统中，互斥量可以解决优先级翻转问题，实现的是优先级继承算法。优先级继承是通过在线程 A 尝试获取共享资源而被挂起的期间内，将线程 C 的优先级提升到线程 A 的优先级别，从而解决优先级翻转引起的问题。这样能够防止 C（间接地防止 A）被 B 抢占，如下图所示。优先级继承是指，提高某个占有某种资源的低优先级线程的优先级，使之与所有等待该资源的线程中优先级最高的那个线程的优先级相等，然后执行，而当这个低优先级线程释放该资源时，优先级重新回到初始设定。因此，继承优先级的线程避免了系统资源被任何中间优先级的线程抢占。

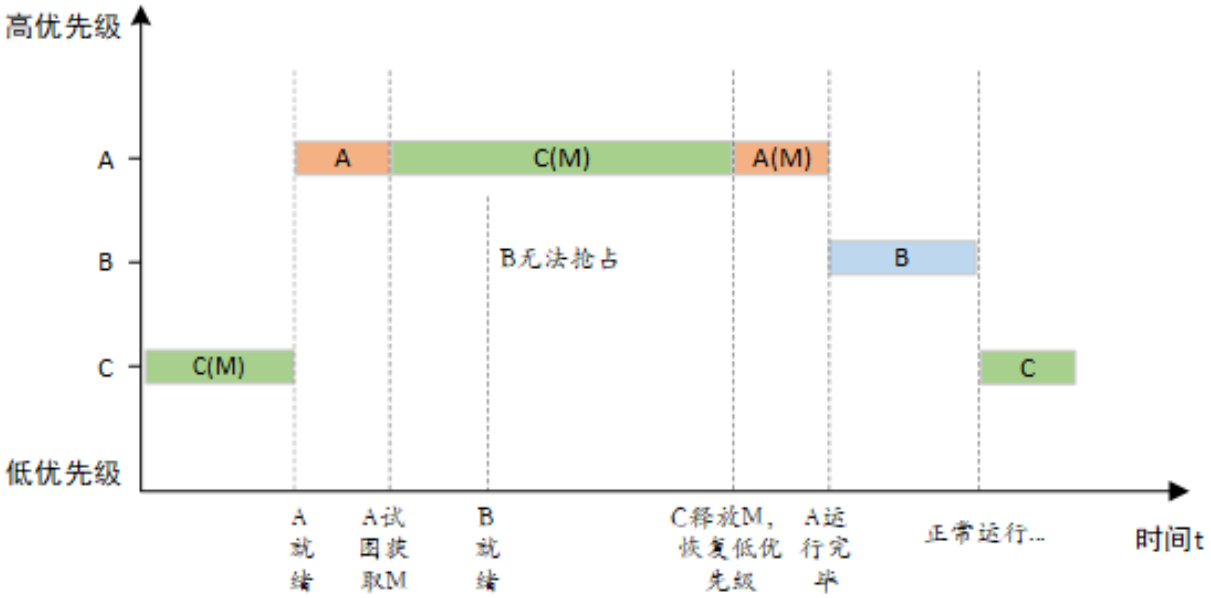

图 8: *优先级继承 (M 为互斥量)*

注意：在获得互斥量后，请尽快释放互斥量，并且在持有互斥量的过程中，不得再行更改持有互斥量线程的优先级。

2.2 互斥量控制块

在 RT-Thread 中，互斥量控制块是操作系统用于管理互斥量的一个数据结构，由结构体 struct rt_mutex 表示。另外一种 C 表达方式 rt_mutex_t，表示的是互斥量的句柄，在 C 语言中的实现是指互斥量控制块的指针。互斥量控制块结构的详细定义请见以下代码：

```
struct rt_mutex
    {
        struct rt_ipc_object parent;                /* 继承自 ipc_object 类 */

        rt_uint16_t          value;                   /* 互斥量的值 */
        rt_uint8_t           original_priority;     /* 持有线程的原始优先级 */
        rt_uint8_t           hold;                     /* 持有线程的持有次数    */
        struct rt_thread    *owner;                 /* 当前拥有互斥量的线程 */
    };
    /* rt_mutext_t 为指向互斥量结构体的指针类型  */
    typedef struct rt_mutex* rt_mutex_t;
```

rt_mutex 对象从 rt_ipc_object 中派生，由 IPC 容器所管理。

2.3 互斥量的管理方式

互斥量控制块中含有互斥相关的重要参数，在互斥量功能的实现中起到重要的作用。互斥量相关接口如下图所示，对一个互斥量的操作包含：创建 / 初始化互斥量、获取互斥量、释放互斥量、删除 / 脱离互斥量。

图 9: *互斥量相关接口*

2.3.1. 创建和删除互斥量

创建一个互斥量时，内核首先创建一个互斥量控制块，然后完成对该控制块的初始化工作。创建互斥量使用下面的函数接口：

```
rt_mutex_t rt_mutex_create (const char* name, rt_uint8_t flag);
```

可以调用 rt_mutex_create 函数创建一个互斥量，它的名字由 name 所指定。当调用这个函数时，系统将先从对象管理器中分配一个 mutex 对象，并初始化这个对象，然后初始化父类 IPC 对象以及与 mutex 相关的部分。互斥量的 flag 标志设置为 RT_IPC_FLAG_PRIO，表示在多个线程等待资源时，将由优先级高的线程优先获得资源。flag 设置为 RT_IPC_FLAG_FIFO，表示在多个线程等待资源时，将按照先来先得的顺序获得资源。下表描述了该函数的输入参数与返回值：

rt_mutex_create() 的输入参数和返回值

参数	描述
name	互斥量的名称
flag	互斥量标志，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	——
互斥量句柄	创建成功
RT_NULL	创建失败

当不再使用互斥量时，通过删除互斥量以释放系统资源，适用于动态创建的互斥量。删除互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_delete (rt_mutex_t mutex);
```

当删除一个互斥量时，所有等待此互斥量的线程都将被唤醒，等待线程获得的返回值是 - RT_ERROR。然后系统将该互斥量从内核对象管理器链表中删除并释放互斥量占用的内存空间。下表描述了该函数的输入参数与返回值：

rt_mutex_delete() 的输入参数和返回值

参数	描述
mutex	互斥量对象的句柄
返回	——
RT_EOK	删除成功

2.3.2. 初始化和脱离互斥量

静态互斥量对象的内存是在系统编译时由编译器分配的，一般放于读写数据段或未初始化数据段中。在使用这类静态互斥量对象前，需要先进行初始化。初始化互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_init (rt_mutex_t mutex, const char* name, rt_uint8_t flag);
```

使用该函数接口时，需指定互斥量对象的句柄（即指向互斥量控制块的指针），互斥量名称以及互斥量标志。互斥量标志可用上面创建互斥量函数里提到的标志。下表描述了该函数的输入参数与返回值：

rt_mutex_init() 的输入参数和返回值

参数	描述
mutex	互斥量对象的句柄，它由用户提供，并指向互斥量对象的内存块
name	互斥量的名称
flag	该标志已经作废，无论用户选择 RT_IPC_FLAG_PRIO 还是 RT_IPC_FLAG_FIFO，内核均按照 RT_IPC_FLAG_PRIO 处理
返回	——
RT_EOK	初始化成功

脱离互斥量将把互斥量对象从内核对象管理器中脱离，适用于静态初始化的互斥量。脱离互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_detach (rt_mutex_t mutex);
```

使用该函数接口后，内核先唤醒所有挂在该互斥量上的线程（线程的返回值是 -RT_ERROR），然后系统将该互斥量从内核对象管理器中脱离。下表描述了该函数的输入参数与返回值：

rt_mutex_detach() 的输入参数和返回值

参数	描述
mutex	互斥量对象的句柄
返回	——
RT_EOK	成功

2.3.3. 获取互斥量

线程获取了互斥量，那么线程就有了对该互斥量的所有权，即某一个时刻一个互斥量只能被一个线程持有。获取互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_take (rt_mutex_t mutex, rt_int32_t time);
```

如果互斥量没有被其他线程控制，那么申请该互斥量的线程将成功获得该互斥量。如果互斥量已经被当前线程线程控制，则该互斥量的持有计数加 1，当前线程也不会挂起等待。如果互斥量已经被其他线程占有，则当前线程在该互斥量上挂起等待，直到其他线程释放它或者等待时间超过指定的超时时间。下表描述了该函数的输入参数与返回值：

rt_mutex_take() 的输入参数和返回值

参数	描述
mutex	互斥量对象的句柄
time	指定等待的时间
返回	——

参数	描述
RT_EOK	成功获得互斥量
-RT_ETIMEOUT	超时
-RT_ERROR	获取失败

2.3.4. 无等待获取互斥量

当用户不想在申请的互斥量上挂起线程进行等待时，可以使用无等待方式获取互斥量，无等待获取互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_trytake(rt_mutex_t mutex);
```

这个函数与 `rt_mutex_take(mutex, RT_WAITING_NO)` 的作用相同，即当线程申请的互斥量资源实例不可用的时候，它不会等待在该互斥量上，而是直接返回 - RT_ETIMEOUT。下表描述了该函数的输入参数与返回值：

参数	描述
mutex	互斥量对象的句柄
返回	——
RT_EOK	成功获得互斥量
-RT_ETIMEOUT	获取失败

2.3.5. 释放互斥量

当线程完成互斥资源的访问后，应尽快释放它占据的互斥量，使得其他线程能及时获取该互斥量。释放互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_release(rt_mutex_t mutex);
```

使用该函数接口时，只有已经拥有互斥量控制权的线程才能释放它，每释放一次该互斥量，它的持有计数就减 1。当该互斥量的持有计数为零时（即持有线程已经释放所有的持有操作），它变为可用，等待在该信号量上的线程将被唤醒。如果线程的运行优先级被互斥量提升，那么当互斥量被释放后，线程恢复为持有互斥量前的优先级。下表描述了该函数的输入参数与返回值：

rt_mutex_release() 的输入参数和返回值

参数	描述
mutex	互斥量对象的句柄
返回	——
RT_EOK	成功

2.4 互斥量应用示例

这是一个互斥量的应用例程，互斥锁是一种保护共享资源的方法。当一个线程拥有互斥锁的时候，可以保护共享资源不被其他线程破坏。下面用一个例子来说明，有两个线程：线程 1 和线程 2，线程 1 对 2 个 number 分别进行加 1 操作；线程 2 也对 2 个 number 分别进行加 1 操作，使用互斥量保证线程改变 2 个 number 值的操作不被打断。如下代码所示：

互斥量例程

```
#include <rtthread.h>

#define THREAD_PRIORITY         8
#define THREAD_TIMESLICE        5

/* 指向互斥量的指针 */
static rt_mutex_t dynamic_mutex = RT_NULL;
static rt_uint8_t number1,number2 = 0;

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;
static void rt_thread_entry1(void *parameter)
{
      while(1)
      {
          /* 线程 1 获取到互斥量后，先后对 number1、number2 进行加 1 操作，然后释放
              互斥量 */
          rt_mutex_take(dynamic_mutex, RT_WAITING_FOREVER);
          number1++;
          rt_thread_mdelay(10);
          number2++;
          rt_mutex_release(dynamic_mutex);
       }
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
static void rt_thread_entry2(void *parameter)
{
      while(1)
      {
          /* 线程 2 获取到互斥量后，检查 number1、number2 的值是否相同，相同则表示
              mutex 起到了锁的作用 */
          rt_mutex_take(dynamic_mutex, RT_WAITING_FOREVER);
          if(number1 != number2)
          {
            rt_kprintf("not protect.number1 = %d, mumber2 = %d \n",number1 ,number2)
                ;
          }
```

```
            else
            {
              rt_kprintf("mutex protect ,number1 = mumber2 is %d\n",number1);
            }

             number1++;
             number2++;
             rt_mutex_release(dynamic_mutex);

            if(number1>=50)
                return;
        }
}

/* 互斥量示例的初始化 */
int mutex_sample(void)
{
    /* 创建一个动态互斥量 */
    dynamic_mutex = rt_mutex_create("dmutex", RT_IPC_FLAG_PRIO);
    if (dynamic_mutex == RT_NULL)
    {
        rt_kprintf("create dynamic mutex failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
                   "thread1",
                   rt_thread_entry1,
                   RT_NULL,
                   &thread1_stack[0],
                   sizeof(thread1_stack),
                   THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,
                   "thread2",
                   rt_thread_entry2,
                   RT_NULL,
                   &thread2_stack[0],
                   sizeof(thread2_stack),
                   THREAD_PRIORITY-1, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);
    return 0;
}

/* 导出到 MSH 命令列表中 */
MSH_CMD_EXPORT(mutex_sample, mutex sample);
```

线程 1 与线程 2 中均使用互斥量保护对 2 个 number 的操作（倘若将线程 1 中的获取、释放互斥量语

句注释掉，线程 1 将对 number 不再做保护），仿真运行结果如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Aug 24 2018
 2006 - 2018 Copyright by rt-thread team
msh >mutex_sample
msh >mutex protect ,number1 = mumber2 is 1
mutex protect ,number1 = mumber2 is 2
mutex protect ,number1 = mumber2 is 3
mutex protect ,number1 = mumber2 is 4
...
mutex protect ,number1 = mumber2 is 48
mutex protect ,number1 = mumber2 is 49
```

线程使用互斥量保护对两个 number 的操作，使 number 值保持一致。

互斥量的另一个例子见下面的代码，这个例子将创建 3 个动态线程以检查持有互斥量时，持有的线程优先级是否被调整到等待线程优先级中的最高优先级。

防止优先级翻转特性例程

```
#include <rtthread.h>

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;
static rt_mutex_t mutex = RT_NULL;


#define THREAD_PRIORITY       10
#define THREAD_STACK_SIZE     512
#define THREAD_TIMESLICE    5

/* 线程 1 入口 */
static void thread1_entry(void *parameter)
{
    /* 先让低优先级线程运行 */
    rt_thread_mdelay(100);

    /* 此时 thread3 持有 mutex，并且 thread2 等待持有 mutex */

    /* 检查 thread2 与 thread3 的优先级情况 */
    if (tid2->current_priority != tid3->current_priority)
    {
        /* 优先级不相同，测试失败 */
        rt_kprintf("the priority of thread2 is: %d\n", tid2->current_priority);
        rt_kprintf("the priority of thread3 is: %d\n", tid3->current_priority);
        rt_kprintf("test failed.\n");
        return;
```

```
    }
    else
    {
        rt_kprintf("the priority of thread2 is: %d\n", tid2->current_priority);
        rt_kprintf("the priority of thread3 is: %d\n", tid3->current_priority);
        rt_kprintf("test OK.\n");
    }
}

/* 线程 2 入口 */
static void thread2_entry(void *parameter)
{
    rt_err_t result;

    rt_kprintf("the priority of thread2 is: %d\n", tid2->current_priority);

    /* 先让低优先级线程运行 */
    rt_thread_mdelay(50);

    /*
     * 试图持有互斥锁，此时 thread3 持有，应把 thread3 的优先级提升
     * 到 thread2 相同的优先级
     */
    result = rt_mutex_take(mutex, RT_WAITING_FOREVER);

    if (result == RT_EOK)
    {
        /* 释放互斥锁 */
        rt_mutex_release(mutex);
    }
}

/* 线程 3 入口 */
static void thread3_entry(void *parameter)
{
    rt_tick_t tick;
    rt_err_t result;

    rt_kprintf("the priority of thread3 is: %d\n", tid3->current_priority);

    result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
    if (result != RT_EOK)
    {
        rt_kprintf("thread3 take a mutex, failed.\n");
    }

    /* 做一个长时间的循环，500ms */
    tick = rt_tick_get();
    while (rt_tick_get() - tick < (RT_TICK_PER_SECOND / 2)) ;
```

```
    rt_mutex_release(mutex);
}

int pri_inversion(void)
{
    /* 创建互斥锁 */
    mutex = rt_mutex_create("mutex", RT_IPC_FLAG_PRIO);
    if (mutex == RT_NULL)
    {
        rt_kprintf("create dynamic mutex failed.\n");
        return -1;
    }

    /* 创建线程 1 */
    tid1 = rt_thread_create("thread1",
                            thread1_entry,
                            RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
         rt_thread_startup(tid1);

    /* 创建线程 2 */
    tid2 = rt_thread_create("thread2",
                            thread2_entry,
                            RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);

    /* 创建线程 3 */
    tid3 = rt_thread_create("thread3",
                            thread3_entry,
                            RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY + 1, THREAD_TIMESLICE);
    if (tid3 != RT_NULL)
        rt_thread_startup(tid3);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(pri_inversion, prio_inversion sample);
```

仿真运行结果如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Aug 27 2018
 2006 - 2018 Copyright by rt-thread team
msh >pri_inversion
the priority of thread2 is: 10
the priority of thread3 is: 11
the priority of thread2 is: 10
the priority of thread3 is: 10
test OK.
```

例程演示了互斥量的使用方法。线程 3 先持有互斥量，而后线程 2 试图持有互斥量，此时线程 3 的优先级被提升为和线程 2 的优先级相同。

注意：需要切记的是互斥量不能在中断服务例程中使用。

2.5 互斥量的使用场合

互斥量的使用比较单一，因为它是信号量的一种，并且它是以锁的形式存在。在初始化的时候，互斥量永远都处于开锁的状态，而被线程持有的时候则立刻转为闭锁的状态。互斥量更适合于：

（1）线程多次持有互斥量的情况下。这样可以避免同一线程多次递归持有而造成死锁的问题。

（2）可能会由于多线程同步而造成优先级翻转的情况。

3 事件集

事件集也是线程间同步的机制之一，一个事件集可以包含多个事件，利用事件集可以完成一对多，多对多的线程间同步。下面以坐公交为例说明事件，在公交站等公交时可能有以下几种情况：

P1 坐公交去某地，只有一种公交可以到达目的地，等到此公交即可出发。

P1 坐公交去某地，有 3 种公交都可以到达目的地，等到其中任意一辆即可出发。

P1 约另一人 P2 一起去某地，则 P1 必须要等到 “同伴 P2 到达公交站” 与 “公交到达公交站” 两个条件都满足后，才能出发。

这里，可以将 P1 去某地视为线程，将 “公交到达公交站”、“同伴 P2 到达公交站” 视为事件的发生，情况 是特定事件唤醒线程；情况 是任意单个事件唤醒线程；情况 是多个事件同时发生才唤醒线程。

3.1 事件集工作机制

事件集主要用于线程间的同步，与信号量不同，它的特点是可以实现一对多，多对多的同步。即一个线程与多个事件的关系可设置为：其中任意一个事件唤醒线程，或几个事件都到达后才唤醒线程进行后续的处理；同样，事件也可以是多个线程同步多个事件。这种多个事件的集合可以用一个 32 位无符号整型变量来表示，变量的每一位代表一个事件，线程通过 “逻辑与” 或 “逻辑或” 将一个或多个事件关联起来，形成事件组合。事件的 “逻辑或” 也称为是独立型同步，指的是线程与任何事件之一发生同步；事件 “逻辑与” 也称为是关联型同步，指的是线程与若干事件都发生同步。

RT-Thread 定义的事件集有以下特点：

1）事件只与线程相关，事件间相互独立：每个线程可拥有 32 个事件标志，采用一个 32 bit 无符号整型数进行记录，每一个 bit 代表一个事件；

2）事件仅用于同步，不提供数据传输功能；

3）事件无排队性，即多次向线程发送同一事件 (如果线程还未来得及读走)，其效果等同于只发送一次。

在 RT-Thread 中，每个线程都拥有一个事件信息标记，它有三个属性，分别是 RT_EVENT_FLAG_AND(逻辑与)，RT_EVENT_FLAG_OR(逻辑或）以及 RT_EVENT_FLAG_CLEAR(清除标记）。当线程等待事件同步时，可以通过 32 个事件标志和这个事件信息标记来判断当前接收的事件是否满足同步条件。

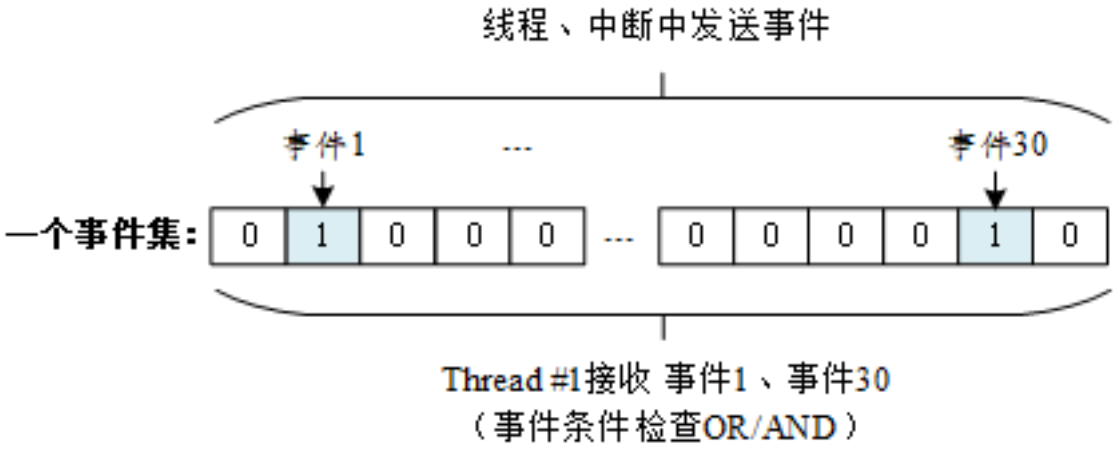

图 10: *事件集工作示意图*

如上图所示，线程 #1 的事件标志中第 1 位和第 30 位被置位，如果事件信息标记位设为逻辑与，则表示线程 #1 只有在事件 1 和事件 30 都发生以后才会被触发唤醒，如果事件信息标记位设为逻辑或，则事件 1 或事件 30 中的任意一个发生都会触发唤醒线程 #1。如果信息标记同时设置了清除标记位，则当线程 #1 唤醒后将主动把事件 1 和事件 30 清为零，否则事件标志将依然存在（即置 1）。

3.2 事件集控制块

在 RT-Thread 中，事件集控制块是操作系统用于管理事件的一个数据结构，由结构体 struct rt_event 表示。另外一种 C 表达方式 rt_event_t，表示的是事件集的句柄，在 C 语言中的实现是事件集控制块的指针。事件集控制块结构的详细定义请见以下代码：

```
struct rt_event
{
    struct rt_ipc_object parent;    /* 继承自 ipc_object 类 */

    /* 事件集合，每一 bit 表示 1 个事件，bit 位的值可以标记某事件是否发生 */
    rt_uint32_t set;
};
/* rt_event_t 是指向事件结构体的指针类型 */
typedef struct rt_event* rt_event_t;
```

rt_event 对象从 rt_ipc_object 中派生，由 IPC 容器所管理。

3.3 事件集的管理方式

事件集控制块中含有与事件集相关的重要参数，在事件集功能的实现中起重要的作用。事件集相关接口如下图所示，对一个事件集的操作包含：创建 / 初始化事件集、发送事件、接收事件、删除 / 脱离事件集。

图 11: *事件相关接口*

3.3.1. 创建和删除事件集

当创建一个事件集时，内核首先创建一个事件集控制块，然后对该事件集控制块进行基本的初始化，创建事件集使用下面的函数接口：

```
rt_event_t rt_event_create(const char* name, rt_uint8_t flag);
```

调用该函数接口时，系统会从对象管理器中分配事件集对象，并初始化这个对象，然后初始化父类 IPC 对象。下表描述了该函数的输入参数与返回值：

rt_event_create() 的输入参数和返回值

参数	描述
name	事件集的名称
flag	事件集的标志，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	——
RT_NULL	创建失败
事件对象的句柄	创建成功

> 注：RT_IPC_FLAG_FIFO 属于非实时调度方式，除非应用程序非常在意先来后到，并且你清楚地明白所有涉及到该事件集的线程都将会变为非实时线程，方可使用 RT_IPC_FLAG_FIFO，否则建议采用 RT_IPC_FLAG_PRIO，即确保线程的实时性。

系统不再使用 rt_event_create() 创建的事件集对象时，通过删除事件集对象控制块来释放系统资源。

删除事件集可以使用下面的函数接口：

```
rt_err_t rt_event_delete(rt_event_t event);
```

在调用 rt_event_delete 函数删除一个事件集对象时，应该确保该事件集不再被使用。在删除前会唤醒所有挂起在该事件集上的线程（线程的返回值是 - RT_ERROR），然后释放事件集对象占用的内存块。下表描述了该函数的输入参数与返回值：

rt_event_delete() 的输入参数和返回值

参数	描述
event	事件集对象的句柄
返回	——
RT_EOK	成功

3.3.2. 初始化和脱离事件集

静态事件集对象的内存是在系统编译时由编译器分配的，一般放于读写数据段或未初始化数据段中。在使用静态事件集对象前，需要先行对它进行初始化操作。初始化事件集使用下面的函数接口：

```
rt_err_t rt_event_init(rt_event_t event, const char* name, rt_uint8_t flag);
```

调用该接口时，需指定静态事件集对象的句柄（即指向事件集控制块的指针），然后系统会初始化事件集对象，并加入到系统对象容器中进行管理。下表描述了该函数的输入参数与返回值：

rt_event_init() 的输入参数和返回值

参数	描述
event	事件集对象的句柄
name	事件集的名称
flag	事件集的标志，它可以取如下数值：RT_IPC_FLAG_FIFO 或 RT_IPC_FLAG_PRIO
返回	——
RT_EOK	成功

系统不再使用 rt_event_init() 初始化的事件集对象时，通过脱离事件集对象控制块来释放系统资源。脱离事件集是将事件集对象从内核对象管理器中脱离。脱离事件集使用下面的函数接口：

```
rt_err_t rt_event_detach(rt_event_t event);
```

用户调用这个函数时，系统首先唤醒所有挂在该事件集等待队列上的线程（线程的返回值是 - RT_ERROR），然后将该事件集从内核对象管理器中脱离。下表描述了该函数的输入参数与返回值：

rt_event_detach() 的输入参数和返回值

参数	描述
event	事件集对象的句柄
返回	——
RT_EOK	成功

3.3.3. 发送事件

发送事件函数可以发送事件集中的一个或多个事件，如下：

```
rt_err_t rt_event_send(rt_event_t event, rt_uint32_t set);
```

使用该函数接口时，通过参数 set 指定的事件标志来设定 event 事件集对象的事件标志值，然后遍历等待在 event 事件集对象上的等待线程链表，判断是否有线程的事件激活要求与当前 event 对象事件标志值匹配，如果有，则唤醒该线程。下表描述了该函数的输入参数与返回值：

rt_event_send() 的输入参数和返回值

参数	描述
event	事件集对象的句柄
set	发送的一个或多个事件的标志值
返回	——
RT_EOK	成功

3.3.4. 接收事件

内核使用 32 位的无符号整数来标识事件集，它的每一位代表一个事件，因此一个事件集对象可同时等待接收 32 个事件，内核可以通过指定选择参数 “逻辑与” 或 “逻辑或” 来选择如何激活线程，使用 “逻辑与” 参数表示只有当所有等待的事件都发生时才激活线程，而使用 “逻辑或” 参数则表示只要有一个等待的事件发生就激活线程。接收事件使用下面的函数接口：

```
rt_err_t rt_event_recv(rt_event_t event,
                       rt_uint32_t set,
                       rt_uint8_t option,
                       rt_int32_t timeout,
                       rt_uint32_t* recved);
```

当用户调用这个接口时，系统首先根据 set 参数和接收选项 option 来判断它要接收的事件是否发生，如果已经发生，则根据参数 option 上是否设置有 RT_EVENT_FLAG_CLEAR 来决定是否重置事件的相应标志位，然后返回（其中 recved 参数返回接收到的事件）；如果没有发生，则把等待的 set 和 option 参数填入线程本身的结构中，然后把线程挂起在此事件上，直到其等待的事件满足条件或等待时间超过指定的超时时间。如果超时时间设置为零，则表示当线程要接受的事件没有满足其要求时就不等待，而直接返回 -RT_ETIMEOUT。下表描述了该函数的输入参数与返回值：

rt_event_recv() 的输入参数和返回值

参数	描述
event	事件集对象的句柄
set	接收线程感兴趣的事件
option	接收选项
timeout	指定超时时间
recved	指向接收到的事件
返回	——
RT_EOK	成功
-RT_ETIMEOUT	超时
-RT_ERROR	错误

option 的值可取：

```
/* 选择 逻辑与 或 逻辑或 的方式接收事件 */
RT_EVENT_FLAG_OR
RT_EVENT_FLAG_AND

/* 选择清除重置事件标志位 */
RT_EVENT_FLAG_CLEAR
```

3.4 事件集应用示例

这是事件集的应用例程，例子中初始化了一个事件集，两个线程。一个线程等待自己关心的事件发生，另外一个线程发送事件，如以下代码所示：

事件集的使用例程

```
#include <rtthread.h>

#define THREAD_PRIORITY      9
#define THREAD_TIMESLICE     5

#define EVENT_FLAG3 (1 << 3)
#define EVENT_FLAG5 (1 << 5)

/* 事件控制块 */
static struct rt_event event;

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;
```

```
/* 线程 1 入口函数 */
static void thread1_recv_event(void *param)
{
    rt_uint32_t e;

    /* 第一次接收事件，事件 3 或事件 5 任意一个可以触发线程 1，接收完后清除事件标志
       */
    if (rt_event_recv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                      RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
                      RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: OR recv event 0x%x\n", e);
    }

    rt_kprintf("thread1: delay 1s to prepare the second event\n");
    rt_thread_mdelay(1000);

    /* 第二次接收事件，事件 3 和事件 5 均发生时才可以触发线程 1，接收完后清除事件标
       志 */
    if (rt_event_recv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                      RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                      RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: AND recv event 0x%x\n", e);
    }
    rt_kprintf("thread1 leave.\n");
}


ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_send_event(void *param)
{
    rt_kprintf("thread2: send event3\n");
    rt_event_send(&event, EVENT_FLAG3);
    rt_thread_mdelay(200);

    rt_kprintf("thread2: send event5\n");
    rt_event_send(&event, EVENT_FLAG5);
    rt_thread_mdelay(200);

    rt_kprintf("thread2: send event3\n");
    rt_event_send(&event, EVENT_FLAG3);
    rt_kprintf("thread2 leave.\n");
}
```

```
int event_sample(void)
{
    rt_err_t result;

    /* 初始化事件对象 */
    result = rt_event_init(&event, "event", RT_IPC_FLAG_PRIO);
    if (result != RT_EOK)
    {
        rt_kprintf("init event failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
                   "thread1",
                   thread1_recv_event,
                   RT_NULL,
                   &thread1_stack[0],
                   sizeof(thread1_stack),
                   THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,
                   "thread2",
                   thread2_send_event,
                   RT_NULL,
                   &thread2_stack[0],
                   sizeof(thread2_stack),
                   THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(event_sample, event sample);
```

仿真运行结果如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Aug 24 2018
 2006 - 2018 Copyright by rt-thread team
msh >event_sample
thread2: send event3
thread1: OR recv event 0x8
thread1: delay 1s to prepare the second event
msh >thread2: send event5
thread2: send event3
thread2 leave.
```

```
thread1: AND recv event 0x28
thread1 leave.
```

例程演示了事件集的使用方法。线程 1 前后两次接收事件，分别使用了“逻辑或”与“逻辑与”的方法。

3.5 事件集的使用场合

事件集可使用于多种场合，它能够在一定程度上替代信号量，用于线程间同步。一个线程或中断服务例程发送一个事件给事件集对象，而后等待的线程被唤醒并对相应的事件进行处理。但是它与信号量不同的是，事件的发送操作在事件未清除前，是不可累计的，而信号量的释放动作是累计的。事件的另一个特性是，接收线程可等待多种事件，即多个事件对应一个线程或多个线程。同时按照线程等待的参数，可选择是“逻辑或”触发还是“逻辑与”触发。这个特性也是信号量等所不具备的，信号量只能识别单一的释放动作，而不能同时等待多种类型的释放。如下图所示为多事件接收示意图：

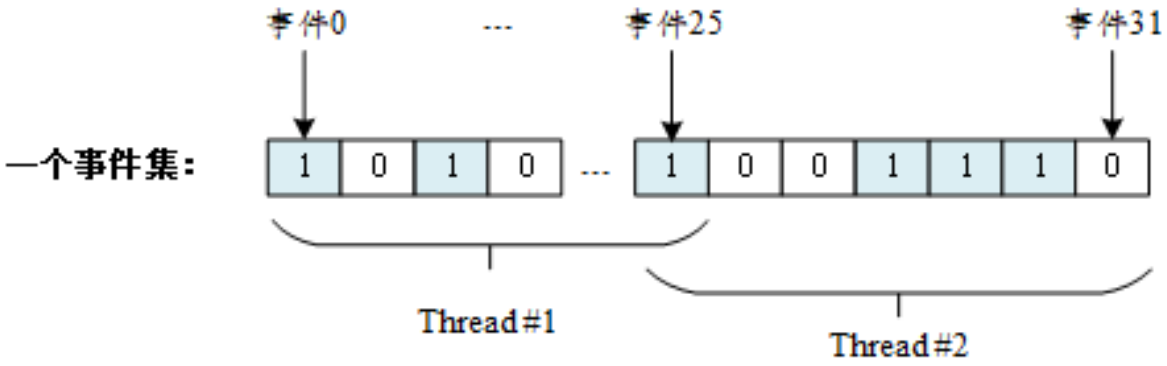

图 12: *多事件接收示意图*

一个事件集中包含 32 个事件，特定线程只等待、接收它关注的事件。可以是一个线程等待多个事件的到来（线程 1、2 均等待多个事件，事件间可以使用“与”或者“或”逻辑触发线程），也可以是多个线程等待一个事件的到来（事件 25）。当有它们关注的事件发生时，线程将被唤醒并进行后续的处理动作。