RT-Thread 任务间同步与通信

Rev: 1.0

2008-9-7

1 任务间同步与通信

在多任务实时系统中，一项工作的完成往往可以通过多个任务来共同合作完成。例如，一个任务从数据采集器中读取数据，然后放到一个链表中进行保存。而另一个任务则从这个链表队列中把数据取出来进行分析处理，并把数据从链表中删除（一个典型的消费者与生产者的例子）。

当消费者任务取到链表的最末端的时候，此时生产者任务可能正在往末端添加数据，此时有可能生产者拿到的末节点被消费者任务给删除了。

正常的操作顺序应该是在一个任务删除或添加动作完成时再进行下一个动作，生产任务与消费任务之间需要协调动作，而对于操作/访问同一块区域，称为临界区。任务的同步方式有很多中，其核心思想是，在访问临界区的时候只允许一个任务运行。

2 关闭中断

关闭中断是禁止多任务访问临界区最简单的一种方式，即使是在分时操作系统中也是如此。当关闭中断的时候，就意味着当前任务不会被其他事件所打断（因为整个系统已经对外部事件不再响应），也就是当前任务不会被抢占，除非这个任务主动退出处理机。

关闭中断/恢复中断 API 是由 BSP 提供。
rt_base_t rt_hw_interrupt_disable() -- 关闭中断
它会返回关闭中断前的中断状态

void rt_hw_interrupt_enable(rt_base_t level) -- 使能中断
它会恢复调用 rt_hw_interrupt_disable 前的中断状态

3 调度器上锁

同样把调度器锁住也能让任务不被换出，直到调度器解锁。但和关闭中断有一点不相同的是，对调度器上锁，系统依然能响应外部中断，中断服务例程依然有可能被运行。所以在使用调度器上锁的方式来做任务同步时，需要考虑好，任务访问的临界资源是否会被中断服务例程所修改。

RT-Thread 提供的调度器操作 API 为：
rt_enter_critical -- 进入临界区

调用这个函数后，调度器将被上锁。在系统锁住调度器的期间，系统依然响应中断，但

因为中断而可能唤醒了高优先级的任务，调度依然不会选择高优先级的任务，直到调用解锁函数才尝试进行下一次调度。

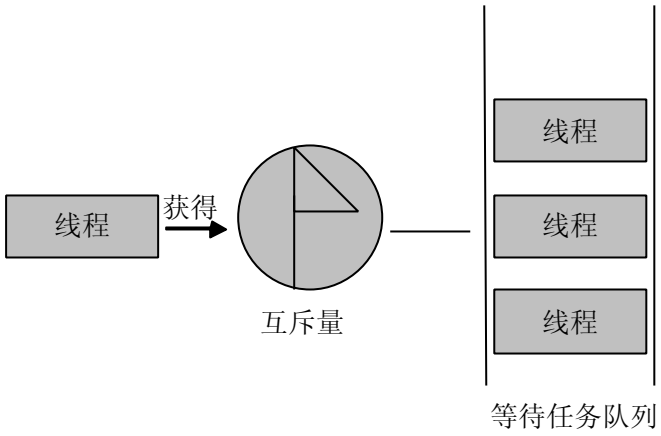
rt_exit_critical -- 退出临界区

在系统退出临界区的时候，系统会计算当前是否有更高优先级的任务就绪，如果有将切换到新的任务中执行，否则继续执行当前任务。

注：rt_enter_critical/rt_exit_critical 可以多次嵌套调用，但有多少次 rt_enter_critical 就必须有成对的多少次 rt_exit_critical。

4 互斥量

互斥量是管理临界资源的一种有效手段。因为互斥量是独占的，所以在一个时刻只允许一个线程占有互斥量，利用这个性质来实现共享资源的互斥量保护。互斥量工作示意图如图 4-5 所示，任何时刻只允许一个线程获得互斥量对象，未能够获得互斥量对象的线程被挂起在该互斥量的等待线程队列上。



使用互斥量会导致的一个潜在问题就是线程优先级翻转。所谓优先级翻转问题即当一个高优先级线程通过互斥量机制访问共享资源时，该互斥量已被一低优先级线程占有，而这个低优先级线程在访问共享资源时可能又被其它一些中等优先级的线程抢先，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。例如：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。

在 RT-Thread 中实现的是优先级继承算法。优先级继承通过在线程 C 被阻塞期间提升线程 A 的优先级到线程 C 的优先级别从而解决优先级翻转引起的问题。这防止了 A（间接地防止 C）被 B 抢占。通俗地说，优先级继承协议使一个拥有资源的线程以等待该资源的线程中优先

级最高的线程的优先级执行。当线程释放该资源时，它将返回到它正常的或标准的优先级。因此，继承优先级的线程避免了被任何中间优先级的线程抢占。

4.1 互斥量控制块

互斥量控制块的数据结构

```
struct rt_mutex
{
struct rt_ipc_object parent;
rt_base_t value;                /* value of mutex. */
struct rt_thread* owner;        /* current owner of mutex. */
rt_uint8_t original_priority;   /* priority of last thread hold the mutex. */
rt_base_t hold;                 /* numbers of thread hold the mutex. */
};
```

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

4.2 互斥量相关接口

4.2.1 创建互斥量

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。创建互斥量使用以下接口：

```
rt_mutex_t rt_mutex_create (const char* name, rt_uint8 flag)
```

使用该接口时，为互斥量指定一个名字，并指定互斥量标志，可以是基于优先级的或基于 FIFO 的。

4.2.2 删除互斥量

系统不再使用互斥量时，通过删除互斥量以释放系统资源。删除互斥量使用以下接口：

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

删除一个互斥量，必须确保该互斥量不再被使用。

4.2.3 初始化互斥量

系统选择静态内存管理方式时，系统会在编译时创建将会使用的各种内核对象，互斥量也会在此时被创建，此时使用互斥量就不再需要使用 rt_mutex_create 接口来创建它，而只需直接对内核在编译时创建的互斥量控制块进行初始化。初始化互斥量使用以下接口：

```
rt_err_t rt_mutex_init (rt_mutex_t mutex, const char* name,
```

```
rt_uint8  flag)
```

使用该接口时，需指定内核已在编译时分配的静态互斥量控制块，指定该互斥量名称以及互斥量标志。

4.2.4 脱离互斥量

脱离互斥量将使互斥量对象被从内核对象管理器中删除。脱离互斥量使用以下接口。

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

使用该接口后，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），然后将该互斥量从内核对象管理器中删除。

4.2.5 获取互斥量

线程通过互斥量申请服务获取对互斥量的控制权。线程对互斥量的控制权是独占的，某一个时刻一个互斥量只能被一个线程控制。在RT-Thread 中使用优先级继承算法来解决优先级翻转问题。成功获得该互斥量的线程的优先级将被提升到等待该互斥量资源的线程中优先级最高的线程的优先级，获取互斥量使用以下接口：

```
rt_err_t  rt_mutex_take  (rt_mutex_t  mutex,  rt_int32  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。如果互斥量已经被当前线程线程控制，则该互斥量的引用计数加一。如果互斥量已经被其他线程控制，则当前线程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。

4.2.6 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。释放互斥量使用以下接口：

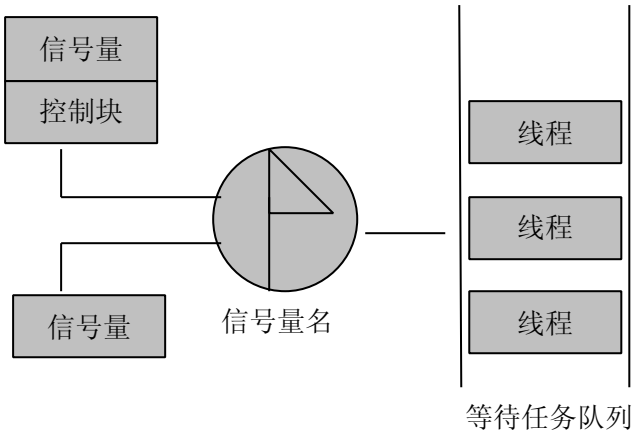
```
rt_err_t  rt_mutex_release(rt_mutex_t  mutex)
```

使用该接口时，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，它的访问计数就减一。当该互斥量的访问计数为零时，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，线程恢复原先的优先级。

5 信号量

信号量是用来解决线程同步和互斥的通用工具，和互斥量类似，信号量也可用作资源互斥访问，但信号量没有所有者的概念，在应用上比互斥量更广泛。信号量比较简单，不能解决优先级翻转问题，但信号量是一种轻量级的对象，比互斥量小巧、灵活。因此在很多对互斥要

求不严格的系统中，经常使用信号量来管理互斥资源。



信号量工作示意图如图 4-6 所示，每个信号量对象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，假如信号量值为 5，则表示共有 5 个信号量实例可以被使用，当信号量实例数目为零时，再申请该信号量的对象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。

5.1 信号量控制块

```
struct rt_semaphore
{
    struct rt_ipc_object parent;

    rt_base_t value;            /* value of semaphore. */
};
```

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

5.2 信号量相关接口

5.2.1 创建信号量

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，创建信号量使用以下接口：

```
rt_sem_t rt_sem_create (const char* name, rt_uint32 value,
rt_uint8 flag);
```

使用该接口时，需为信号量指定一个名称，并指定信号量初始值和信号量标志。

5.2.2 删除信号量

系统不再使用信号量时，通过删除信号量以释放系统资源。删除信号量使用以下接口：

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

删除一个信号量，必须确保该信号量不再被使用。
如果删除该信号量时，有线程正在等待该信号量，则先唤醒等待在该信号量上的线程（返回值为-RT_ERROR）。

5.2.3 初始化信号量

系统选择静态内存管理方式时，系统会在编译时创建将会使用的各种内核对象，信号量也会在此时被创建，此时使用信号量就不再需要使用 rt_mutex_create 接口来创建它，而只需直接对内核在编译时创建的互斥量控制块进行初始化。初始化互斥量使用以下接口：

```
rt_err_t rt_sem_init (rt_sem_t sem, const char* name, rt_uint32 value, rt_uint8 flag)
```

使用该接口时，需指定内核分配的静态信号量控制块，指定信号量名称以及信号量标志。

5.2.4 脱离信号量

脱离信号量将使信号量对象被从内核对象管理器中删除。脱离信号量使用以下接口。

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

使用该接口后，内核先唤醒所有挂在该信号量上的线程，然后将该信号量从内核对象管理器中删除。

5.2.5 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，它每次被申请获得，值都会减一，获取信号量使用以下接口：

```
rt_err_t rt_sem_take (rt_sem_t sem, rt_int32 time)
```

如果信号量的值等于零，那么说明当前资源不可用，申请该信号量的线程就必须在此信号量上等待，直到其他线程释放该信号量或者等待时间超过指定超时时间。

5.2.6 获取无等待信号量

当用户不想在申请的信号量上等待时，可以使用无等待信号量，获取无等待信号量使用以下接口：

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

跟获取信号量接口不同的是，当线程申请的信号量资源不可用的时候，它不是等待在该信号量上，而是直接返回-RT_ETIMEOUT。

5.2.7 释放信号量

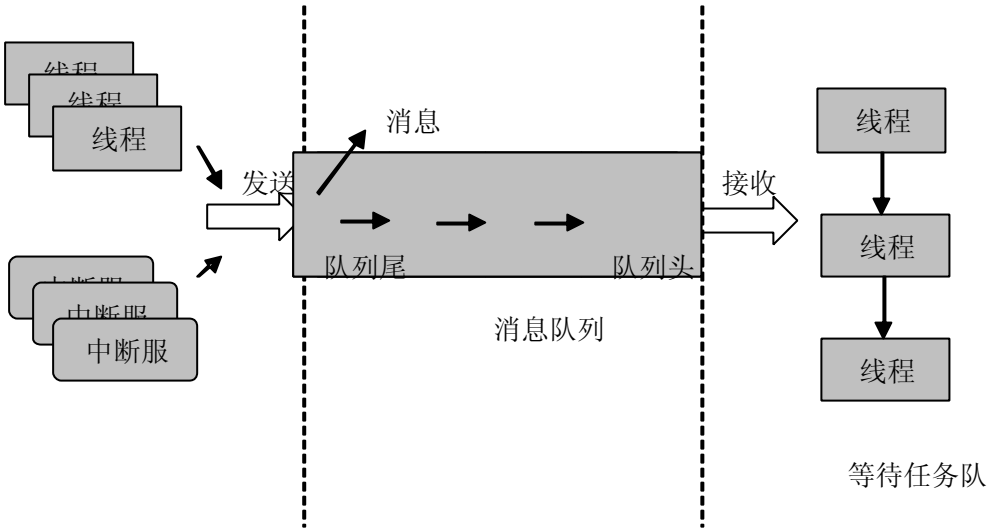
当线程完成信号量资源的访问后，应尽快释放它占据的信号量，使得其他线程能获得该信号量。释放信号量使用以下接口：

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

当信号量的值等于零时，信号量值加一，并且唤醒等待在该信号量上的线程队列中的首线程，由它获取信号量。

6 消息队列

消息队列是另一种常用的线程间通讯方式，它使得线程之间接收和发送消息不必同时进行，消息队列也有点类似管道，它临时保存线程从队列的一端发送来的消息，直到有接收者读取它们。



消息队列用于给线程发消息。如图 4-13 所示，通过内核提供的服务，线程或中断服务子程序可以将一条消息放入消息队列。同样，一个或多个线程可以通过内核服务从消息队列中得到消息。通常，先进入消息队列的消息先传给线程，也就是说，线程先得到的是最先进入消息队列的消息，即先进先出原则(FIFO)。

消息队列由多个元素组成，内核用它们来管理队列。当消息队列被创建时，它就被分配了消息队列控制块，队列名，内存缓冲区，消息大小以及队列长度。

内核负责给消息队列分配消息队列控制块，它同时也接收用户线程传入的参数，像消息队列名以及消息大小，队列长度，由这些来确定消息队列所需内存大小，当获得了这些信息后，内核就可以从内存堆或者线程私有内存空间中为消息队列分配内存。

一个消息队列中本身包含着多个消息框，每个消息框可以存放一条消息，消息队列中的

第一个和最后一个消息框被分别称为队首和队尾，对应了消息队列控制块中的msg_queue_head 和 msg_queue_tail，有些消息框中可能是空的，所有消息队列中的消息框总数就是消息队列的长度。用户线程可以在创建消息队列时指定这个长度。

6.1 消息队列控制块

```
struct rt_messagequeue
{
    struct rt_ipc_object parent;

    void* msg_pool;         /* start address of message queue. */

    rt_size_t msg_size;     /* message size of each message. */
    rt_size_t max_msgs;     /* max number of messages. */

    void* msg_queue_head;   /* list head. */
    void* msg_queue_tail;   /* list tail. */
    void* msg_queue_free;   /* pointer indicated the free node of
queue. */

    rt_ubase_t entry;       /* index of messages in the queue. */
};
```

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

6.2 消息队列相关接口

6.2.1 创建消息队列

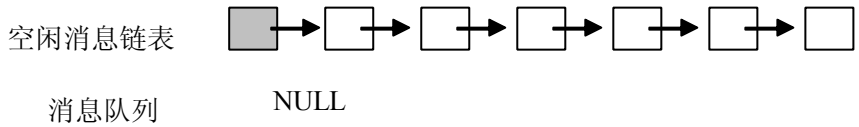


图 3-14 创建消息队列时的内部结构图

创建消息队列时先创建一个消息队列对象控制块，然后给消息队列分配一块内存空间组织成空闲消息链表，这块内存大小等于消息大小与消息队列容量的乘积。然后再初始化消息队列，此时消息队列为空，如图所示。创建消息队列接口如下：

```
rt_mq_t rt_mq_create (const char* name, rt_size_t msg_size,
rt_size_t max_msgs, rt_uint8 flag)
```

创建消息队列时给消息队列指定一个名字，作为消息队列的标识，然后根据用户需求指定消

息的大小以及消息队列的容量。如图 3-14 所示，消息队列被创建时所有消息都挂在空闲消息列表上，消息队列为空。

6.2.2 删除消息队列

当消息队列不再被使用时，应该删除它以释放系统资源，一旦操作完成，消息队列将被永久性的删除。删除消息队列接口如下：

```
rt_err_t  rt_mq_delete  (rt_mq_t mq)
```

删除消息队列时，如果有线程被挂起在该消息队列等待队列上，则先唤醒挂起在该消息等待队列上的所有线程（返回值是-RT_ERROR），然后再释放消息队列使用的内存，最后删除消息队列对象。

6.2.3 脱离消息队列

脱离消息队列将使消息队列对象被从内核对象管理器中删除。脱离消息队列使用以下接口。

```
rt_err_t  rt_mq_detach(rt_mq_t mq)
```

使用该接口后，内核先唤醒所有挂在该消息等待队列对象上的线程（返回值是-RT_ERROR），然后将该消息队列对象从内核对象管理器中删除。

6.2.4 初始化消息队列

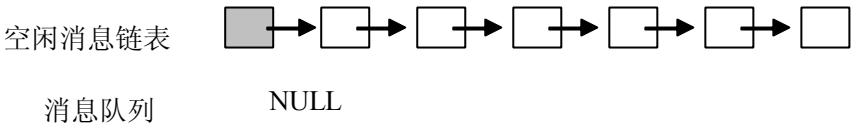


图 4-15 初始化消息队列时的内部结构图

初始化消息队列跟创建消息队列类似，只是初始化消息队列用于静态内存管理模式，消息队列控制块来源于用户线程在系统中申请的静态对象。还与创建消息队列不同的是，此处消息队列对象所使用的内存空间是由用户线程提供的一个缓冲区空间，其余的初始化工作与创建消息队列时相同。初始化消息队列接口如下：

```
rt_err_t  rt_mq_init(rt_mq_t  mq,  const  char*  name,  void  *msgpool,
rt_size_t msg_size, rt_size_t pool_size, rt_uint8 flag)
```

初始化消息队列时，该接口需要获得用户已经申请获得的消息队列控制块以及缓冲区指针参数，以及线程指定的消息队列名，消息大小以及消息队列容量。

如图 4-15 所示，消息队列初始化后所有消息都挂在空闲消息列表上，消息队列为空。

6.2.5 发送消息

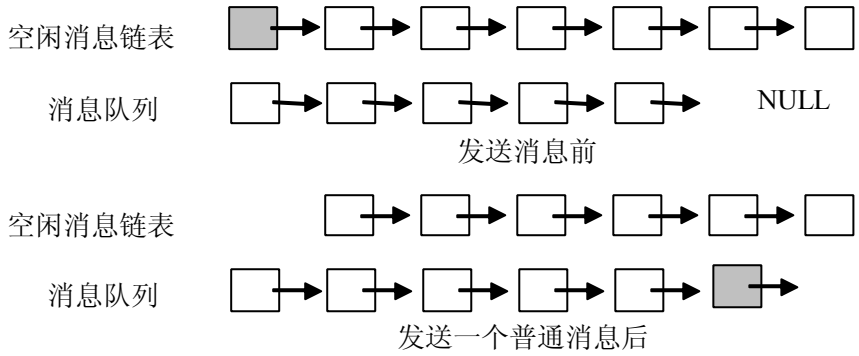


图 3-16 消息队列在发送普通消息后的内部结构图

线程或者中断服务程序都可以给消息队列发送消息，当发送消息时，内核先从空闲消息链表上取下一个空闲消息块，把线程或者中断服务程序发送的消息内容复制到消息块上，然后把该消息块挂到消息队列的尾部。发送者成功发送消息当且仅当空闲消息链表上有可用的空闲消息块；当自由消息链表上无可用消息块，说明消息队列中的消息已满，此时，发送消息的的线程或者中断程序会收到一个错误码。发送消息接口如下：

```
rt_err_t rt_mq_send (rt_mq_t mq, void* buffer, rt_size_t size)
```

发送消息时，发送者需指定发送到哪个消息队列，并且指定发送的消息内容以及消息大小。如图 3-16 所示，在发送一个普通消息之后，空闲消息链表上的队首消息被转移到了消息队列尾。

6.2.6 发送紧急消息

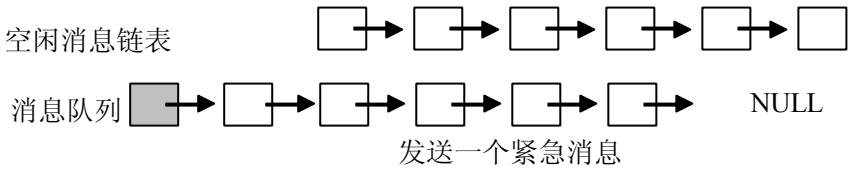


图 4-17 消息队列在发送紧急消息后的内部结构图

发送紧急消息的过程与发送消息几乎一样，唯一的不同是，当发送紧急消息时，从空闲消息链表上取下来的消息块不是挂到消息队列的队尾，而是挂到队首，这样，接收者能够优先接收到紧急消息，从而及时进行消息处理。发送紧急消息的接口如下：

```
rt_err_t rt_mq_urgent(rt_mq_t mq, void* buffer, rt_size_t size)
```

如图 4-17 所示，在发送一个紧急消息之后，空闲消息链表上的队首消息被转移到了消息队列首。

6.2.7 接收消息

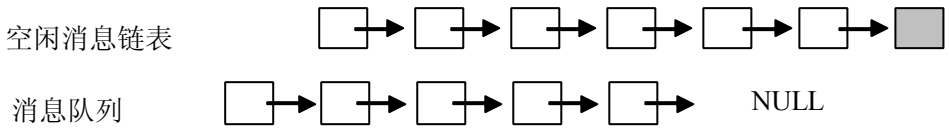


图 4-18 消息队列在接收消息后的内部结构图

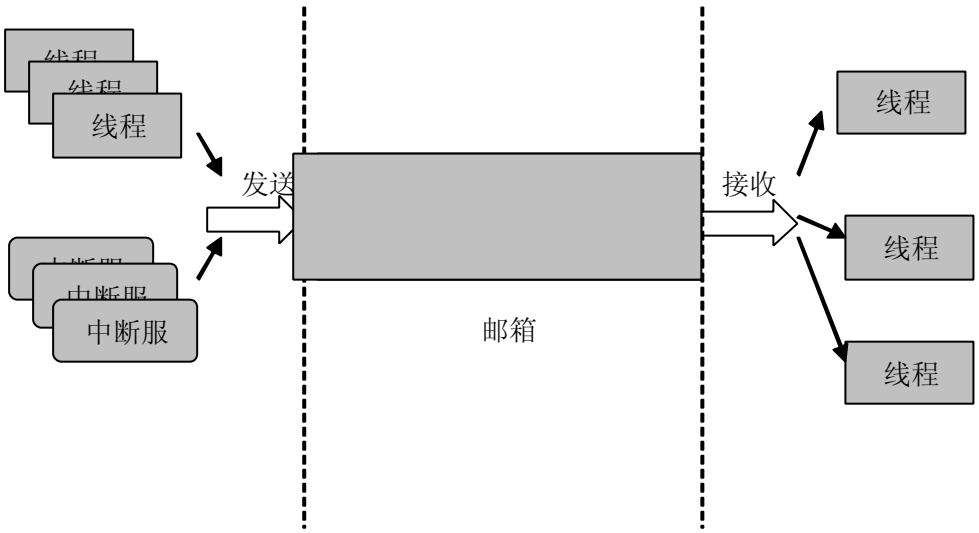
注：只有线程能够接收消息队列中的消息。只有当消息队列中有消息时，接收者才能接收消息，否则接收者会根据超时时间设置或挂起在消息队列的等待线程队列上，或直接返回。接收消息接口如下：

```
rt_err_t rt_mq_recv (rt_mq_t mq, void* buffer, rt_size_t size,
rt_int32 timeout)
```

接收消息时，接收者需指定存储消息的消息队列名,并且指定一个内存缓冲区，接收到的消息内容将被复制到该缓冲区，此外，还需指定未能及时取到邮件时的超时时间。如图 4-18 所示，接收一个消息后消息队列上的队首消息被转移到了空闲消息链表的尾部。

7 邮箱

通过内核服务可以给线程发送消息。典型的邮箱也称作交换消息，如图 4-9 所示，是用一个指针型变量，通过内核服务，一个线程或一个中断服务程序可以把一则消息(即一个指针)放到邮箱里去。同样，一个或多个线程可以通过内核服务接收这则消息。



RT-Thread 采用的邮箱通信机制有点类型传统意义上的管道，用于线程间通讯。它是线程，中断服务，定时器向线程发送消息的有效手段。邮箱与线程对象等是独立的。线程，中断服务和定时器都可以向邮箱发送消息，但是只有线程能够接收消息。

RT-Thread 的邮箱中共可存放固定条数的邮件，邮箱容量在创建邮箱时设定，每个邮件大小为 32 比特，刚好是一个整型数或一个指针大小。当实际需要发送的邮件较大时，可以

传递指向一个缓冲区的指针。邮件格式不受限制，可以是任意格式的，可以是字符串，指针，二进制或其他格式。

当邮箱满时，线程等不再发送新邮件。当邮箱空时，挂起正在试图接收邮件的线程，使其等待，当邮箱中有新邮件时，唤醒等待在邮箱上的线程，使其能够接收新邮件。

7.1 邮箱控制块

```
struct rt_mailbox
{
    struct rt_ipc_object parent;

    rt_uint32_t* msg_pool;  /* start address of message buffer. */

    rt_size_t size;         /* size of message pool. */

    rt_ubase_t entry;       /* index of messages in msg_pool. */
    rt_ubase_t in_offset, out_offset; /* in/output offset of the
message buffer. */
};
```

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

7.2 邮箱相关接口

7.2.1 创建邮箱

创建邮箱对象时先创建一个邮箱对象控制块，然后给邮箱分配一块内存空间用来存放邮件，这块内存大小等于邮件大小与邮箱容量的乘积，接着初始化接收邮件和发送邮件在邮箱中的偏移量。创建邮箱的接口如下：

```
rt_mailbox_t rt_mb_create (const char* name, rt_size_t size,
rt_uint8 flag)
```

创建邮箱时给邮箱指定一个名称，作为邮箱的标识，并且指定邮箱的容量。

7.2.2 删除邮箱

当邮箱不再被使用时，应该删除它以释放系统资源，一旦操作完成，邮箱将被永久性的删除。删除邮箱接口如下：

```
rt_err_t rt_mb_delete (rt_mailbox_t mb)
```

删除邮箱时，如果有线程被挂起在该邮箱对象上，则先唤醒挂起在该邮箱上的所有线程，然后再释放邮箱使用的内存，最后删除邮箱对象。

7.2.3 脱离邮箱

脱离邮箱将使邮箱对象被从内核对象管理器中删除。脱离邮箱使用以下接口。

```
rt_err_t  rt_mb_detach(rt_mailbox_t  mb)
```

使用该接口后，内核先唤醒所有挂在该邮箱上的线程，然后将该邮箱对象从内核对象管理器中删除。

7.2.4 初始化邮箱

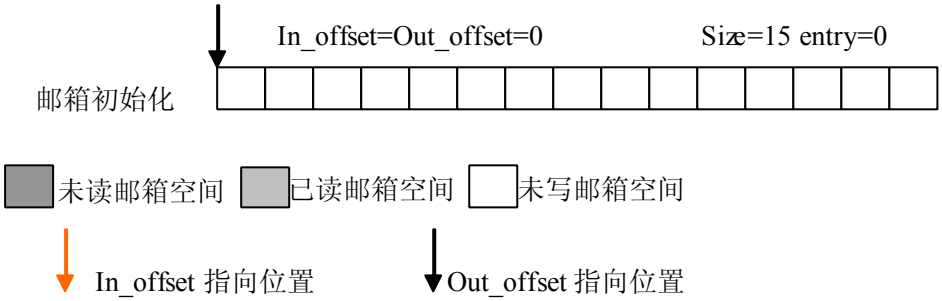


图 4-10 邮箱初始化后的内部结构

初始化邮箱跟创建邮箱类似，只是初始化邮箱用于静态内存管理模式，邮箱控制块来源于用户线程在系统中申请的静态对象。还与创建邮箱不同的是，此处邮箱对象所使用的内存空间是由用户线程指定的一个缓冲区空间，用户把缓冲区的指针传递给邮箱对象控制块，其余的初始化工作与创建邮箱时相同。接口如下：

```
rt_err_t  rt_mb_init(rt_mailbox_t  mb,  const  char*  name,  void*
msgpool,  rt_size_t  size,  rt_uint8  flag)
```

初始化邮箱时，该接口需要获得用户已经申请获得的邮箱对象控制块以及缓冲区指针参数，以及线程指定的邮箱名和邮箱容量。如图 4-10 所示，邮箱初始化后接收邮件偏移量 In_offset，Out_offset 均为零，邮箱容量 size 为 15,邮箱中邮件数目 entry 为 0。

7.2.5 发送邮件

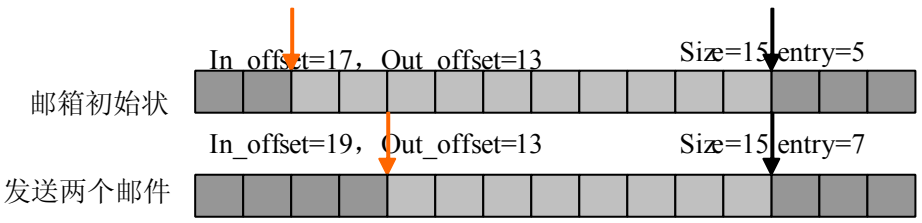


图 4-11 邮箱发送邮件后的内部结构

线程或者中断服务程序通过邮箱可以给其他线程发送邮件，发送的邮件可以是 32 位任意格式的数据，一个整型值或者指向一个缓冲区的指针，发送者成功发送邮件当且仅当邮箱未满，当邮箱中的邮件满时，发送邮件的线程或者中断程序会收到一个错误码。发送邮件接口如下：

```
rt_err_t rt_mb_send (rt_mailbox_t mb, rt_uint32 value)
```

发送邮件发送者需指定接收邮箱名称，并且指定发送的邮件内容。如图 4-11 所示，邮箱容量 size 为 15，发送邮件之前邮件数 entry 为 5，接收邮件偏移量为 17，指向第三个邮件头。发送邮件偏移量为 13，指向第 13 个邮件头。当连续发送两个邮件后，邮箱中邮件数变为 7，接收邮件偏移量为 19，指向第五个邮件头，发送邮件偏移量不变。

7.2.6 接收邮件

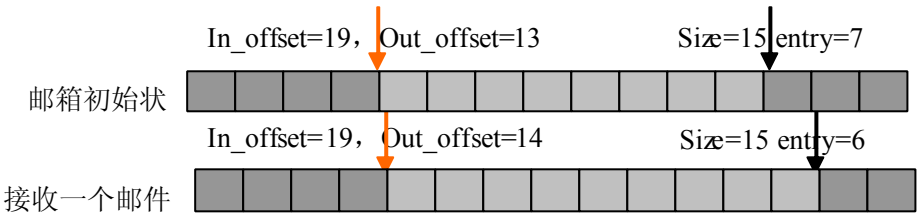


图 4-12 邮箱接收邮件后的内部结构图

只有线程能够接收邮箱中的邮件。只有当接收者接收的邮箱中有邮件时，接收者才能立即取到邮件，否则接收线程会根据超时时间设置或挂起在邮箱的等待线程队列上，或直接返回。接收邮件接口如下：

```
rt_err_t rt_mb_recv (rt_mailbox_t mb, rt_uint32* value, rt_int32
timeout)
```

接收邮件时，接收者需指定接收邮件的邮箱,并指定接收到的邮件存放位置以及未能及时取到邮件时的超时时间。如图 4-12 所示，邮箱容量 size 为 15，发送邮件之前邮件数 entry 为 7，接收邮件偏移量为 19，指向第五个邮件头。发送邮件偏移量为 13，指向第 13 个邮件头。当连续发送两个邮件后，邮箱中邮件数变为 6，接收邮件偏移量不变，接收邮件偏移量变为 14，指向第 14 个邮件头。

8 事件

事件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。可以是一个线程同步多个事件，也可以是多个线程同步多个事件。多个事件的集合用一个无符号整型变量来表示，变量中的一位代表一个事件，线程通过“逻辑与”或“逻辑或”与一个或多个事件建立关联。

往往内核会定义若干个事件，称为事件集。事件在用于同步时有两种类型，一种是独立型同步，是指线程与任何事件之一发生同步（逻辑或关系），另一种是关联型同步，是指线程与若干事件都发生同步（逻辑与关系）。

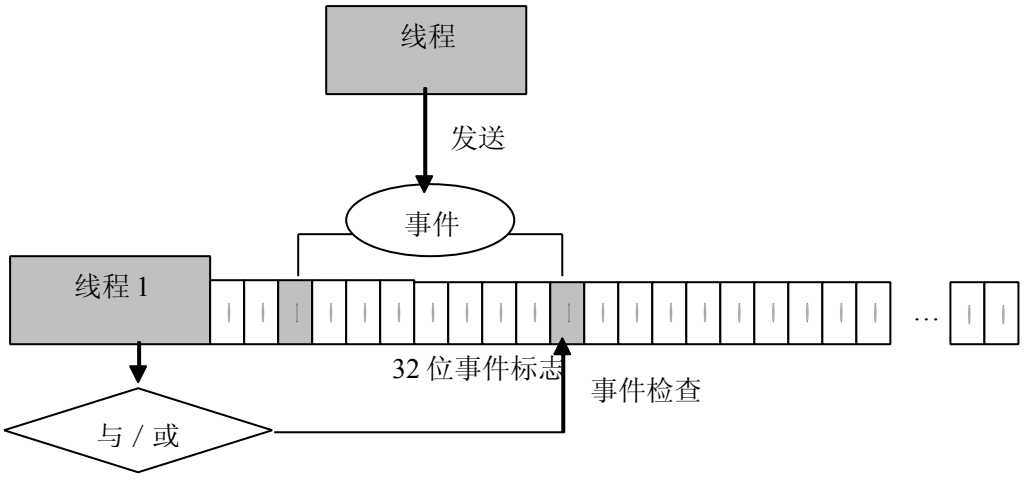


图 4-7 事件对象工作示意图

RT-Thread 定义的事件有以下特点：

1) 事件只与线程相关，事件间相互独立，RT-Thread 定义的每个线程拥有 32 个事件标志，用一个 32-bit 无符号整形数记录，每一个 bit 代表一个事件。若干个事件构成一个事件集。
2) 事件仅用于同步，不提供数据传输功能
3) 事件无队列，即多次向线程发送同一事件，其效果等同于只发送一次。

在 RT-Thread 中，每个线程还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，就可以通过 32 个事件标志和一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。

如图 4-7 所示，线程 1 的事件标志中第三位和第十位被置位，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 3 和事件 10 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 3 或事件 10 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，则发生的事件会导致线程 1 的相应事件标志位被重新置位为零。

8.1 事件控制块

```
struct rt_event
{
    struct rt_ipc_object parent;
    rt_uint32 set;
};
```

8.2 事件相关接口

8.2.1 创建事件

当创建一个事件时，内核首先创建一个事件控制块，然后对该事件控制块进行基本的初始化，

创建事件使用以下接口：

```
rt_event_t rt_event_create (const char* name, rt_uint8 flag)
```

使用该接口时，需为事件指定名称以及事件标志。

8.2.2 删除事件

系统不再使用事件对象时，通过删除事件对象控制块以释放系统资源。删除事件使用以下接口：

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

删除一个事件，必须确保该事件不再被使用，同时唤醒所有挂起在该事件上的线程。

8.2.3 脱离事件

脱离信号量将使事件对象从内核对象管理器中删除。脱离事件使用以下接口。

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

使用该接口后，内核首先唤醒所有挂在该事件上的线程，然后将该事件从内核对象管理器中删除。

8.2.4 初始化事件

选择静态内存管理方式时，在编译时创建将会使用的各种内核对象，事件对象也会在此时被创建，此时使用事件就不再需要使用 rt_event_create 接口来创建它，而只需直接对内核在编译时创建的事件控制块进行初始化。初始化事件使用以下接口：

```
rt_err_t rt_event_init(rt_event_t event, const char* name,
rt_uint8 flag)
```

使用该接口时，需指定内核分配的静态事件对象，并指定事件名称和事件标志。

8.2.5 接收事件

内核使用 32 位的无符号整型数来标识事件，它的每一位代表一个事件，因此一个事件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。接收事件使用以下接口：

```
rt_err_t rt_event_recv(rt_event_t event, rt_uint32 set, rt_uint8
option, rt_int32 timeout, rt_uint32* recved)
```

用户线程首先根据选择参数和事件对象标志来判断它要接收的事件是否发生，如果已经发生，则根据线程要求选择是否重置事件对象的相应标志位，然后返回，如果没有发生，则把

等待的事件标志位和选择参数填入线程本身的结构中，然后把线程挂起在此事件对象上，直到其等待的事件满足条件或等待时间超过指定超时时间。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不等待而直接返回。

8.2.6 发送事件

通过发送事件服务，可以发送一个或多个事件。发送事件使用以下接口：

```
rt_err_t  rt_event_send(rt_event_t  event,  rt_uint32  set)
```

使用该接口时，通过 set 参数指定的事件标志重新设定 event 对象的事件标志值，然后遍历等待在 event 事件上的线程链表，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，则激活该线程。

9 快速事件

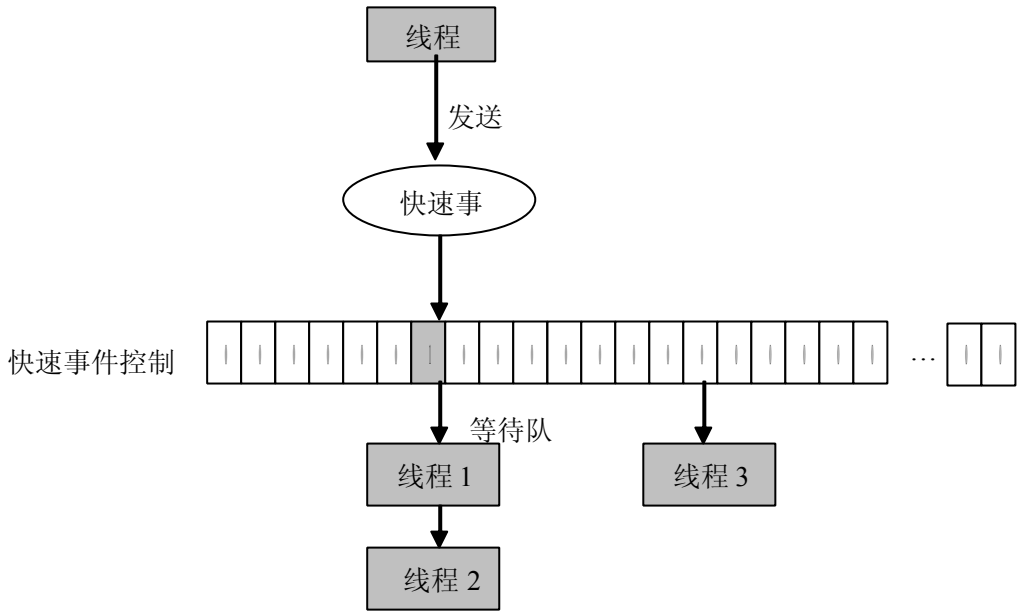


图 4-8 快速事件工作示意图

快速事件对象和事件对象非常类似，快速事件对象也是 32 位整数，一个事件也是一个二进制位，每个事件位会拥有一个线程队列，当事件到来时，直接从队列中获得等待的线程并唤醒[所以寻找等待线程的时间是确定的]。快速事件对于通常的 RTOS 而言优势并不大，但会用于微内核中断处理中。如图 4-8，线程 1，和线程 2 因为等待事件 7 被挂起在事件 7 的线程队列上，线程 3 因为等待事件 15 而被挂在事件 15 的线程队列上，这时，有其他线程发送了一个事件 7，于是线程 1 和线程 2 被从事件 7 的线程队列上唤醒，重新进入就绪队列。

9.1 快速事件控制块

```
struct  rt_fast_event
```

```
{
    struct rt_object parent;
    rt_uint32 set;
    rt_list_t thread_list[RT_EVENT_LENGTH];
};
```

9.2 快速相关接口

9.2.1 创建快速事件

当创建一个快速事件时，内核首先创建一个快速事件控制块，并对该控制块进行必要的初始化。创建快速事件使用以下接口：

```
rt_event_t rt_fast_event_create (const char* name, rt_uint8 flag)
```

使用该接口时，需指定一个事件名称及事件标志。

9.2.2 删除快速事件

系统不再使用快速事件对象时，通过删除事件对象控制块以释放系统资源。删除快速事件使用以下接口：

```
rt_err_t rt_fast_event_delete (rt_fast_event_t event)
```

删除一个快速事件，必须确保该快速事件不再被使用，然后唤醒所有挂起在该快速事件上的线程。

9.2.3 脱离快速事件

脱离信号量将使快速事件对象被从内核对象管理器中删除。脱离快速事件使用以下接口。

```
rt_err_t rt_fast_event_detach(rt_fast_event_t event)
```

使用该接口后，内核先唤醒所有挂在该快速事件对象上的线程，然后将该快速事件对象从内核对象管理器中删除。

9.2.4 初始化快速事件

选择静态内存管理方式时，在编译时创建将会使用的各种内核对象，快速事件对象也会在此时被创建，此时使用事件就不再需要使用 rt_fast_event_create 接口来创建它，而只需直接对内核在编译时创建的快速事件控制块进行初始化。初始化事件使用以下接口：

```
rt_err_t rt_fast_event_init(rt_fast_event_t event, const char*
name, rt_uint8 flag)
```

使用该接口时，需要指定系统中已分配的静态快速事件对象，给该对象指定名称，并传入事件标志。

9.2.5 接收快速事件

系统使用 32 位的无符号整型数来标识快速事件，它的每一位代表一个快速事件，因此一个事件对象可同时等待接收 32 个事件，每个事件都对应有一个线程等待队列，接收快速事件使用以下接口：

```
rt_err_t rt_fast_event_recv(rt_fast_event_t event, rt_uint8 bit, rt_uint8 option, rt_int32 timeout)
```

接收快速事件时，接收者首先根据快速事件对象标志来判断它要接收的事件是否发生，如果已经发生，则根据线程要求选择是否重置事件对象的相应标志位，然后返回，如果没有发生，则将自己挂起在此事件对应的线程等待队列上，直到其等待的事件发生或等待时间超过指定超时时间。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不等待而直接返回。

9.2.6 发送快速事件

通过发送快速事件服务，可以发送一个快速事件。发送快速事件使用以下接口：

```
rt_err_t rt_fast_event_send(rt_fast_event_t event, rt_uint8 bit)
```

发送快速事件时，通过 bit 位来指定发送的事件，内核会遍历等待在该事件对应的等待线程队列，如果该队列上有等待的线程，则依次唤醒这些等待线程。