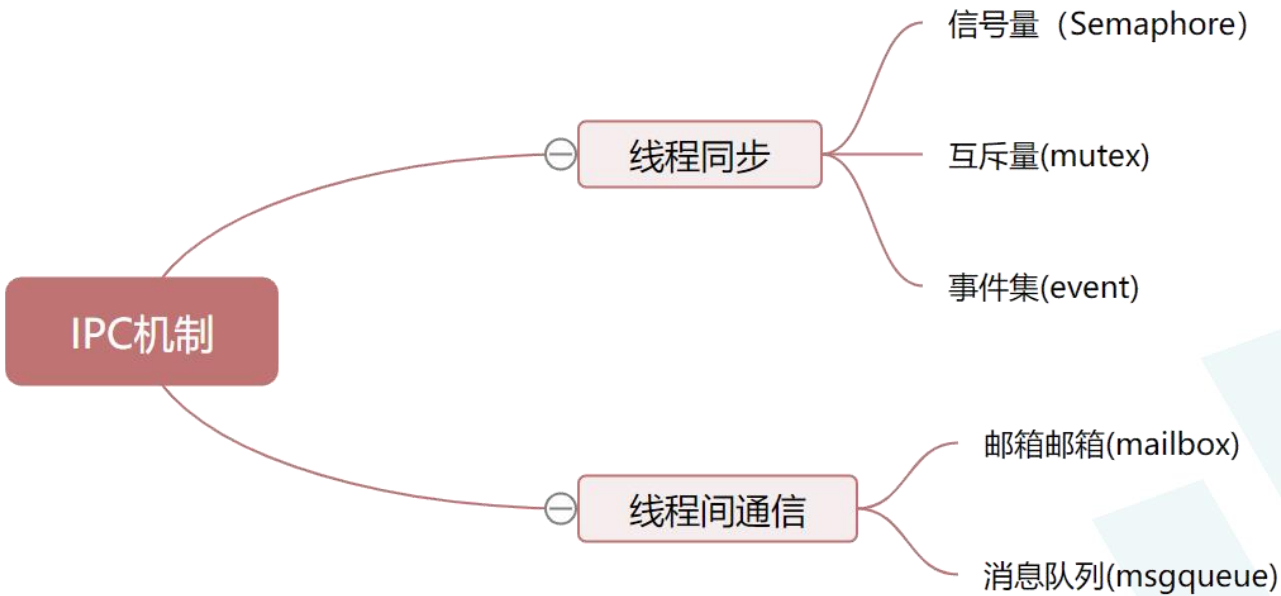
RT-Thread

RT-Thread IPC

RT-Thread RV

2024/07/23

内容概览


IPC机制
线程同步
信号量（Semaphore）
互斥量(mutex)
事件集(event)
线程间通信
邮箱邮箱(mailbox)
消息队列(msgqueue)

课前问题

RTOS与生活中的一些问题的联系

问题1：不认识的俩个人线下见面怎么找到对方？

答案：信号量？

问题2：合租只有一个卫生间，有三个租户怎么协调？

答案：互斥量？信号量可以吗？

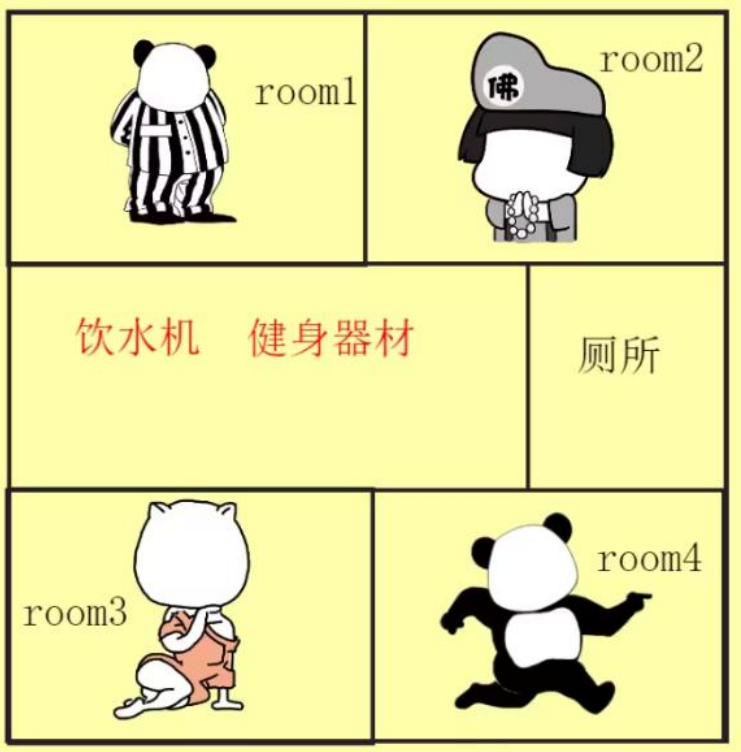
问题3：你想去卫生间的时候，有人在怎么办？

答案：阻塞？非阻塞？挂起？死锁？

RT-Thread

典型概念——临界区

内核入门——典型概念



什么是临界区？

临界区的资源在同一时间只能被一个线程(住户)使用，所以一旦临界资源被占用，其他的线程(住户)能做的就只有等待。

RT-Thread

典型概念——阻塞与非阻塞

内核入门——典型概念

了解了临界区的概念之后，阻塞概念就好理解了。一个线程先占用了临界区的资源，此时如果其他的线程想使用临界区资源就必须等待。这种占用临界区资源，阻塞其他线程继续执行的情况就是线程阻塞（Blocking）。

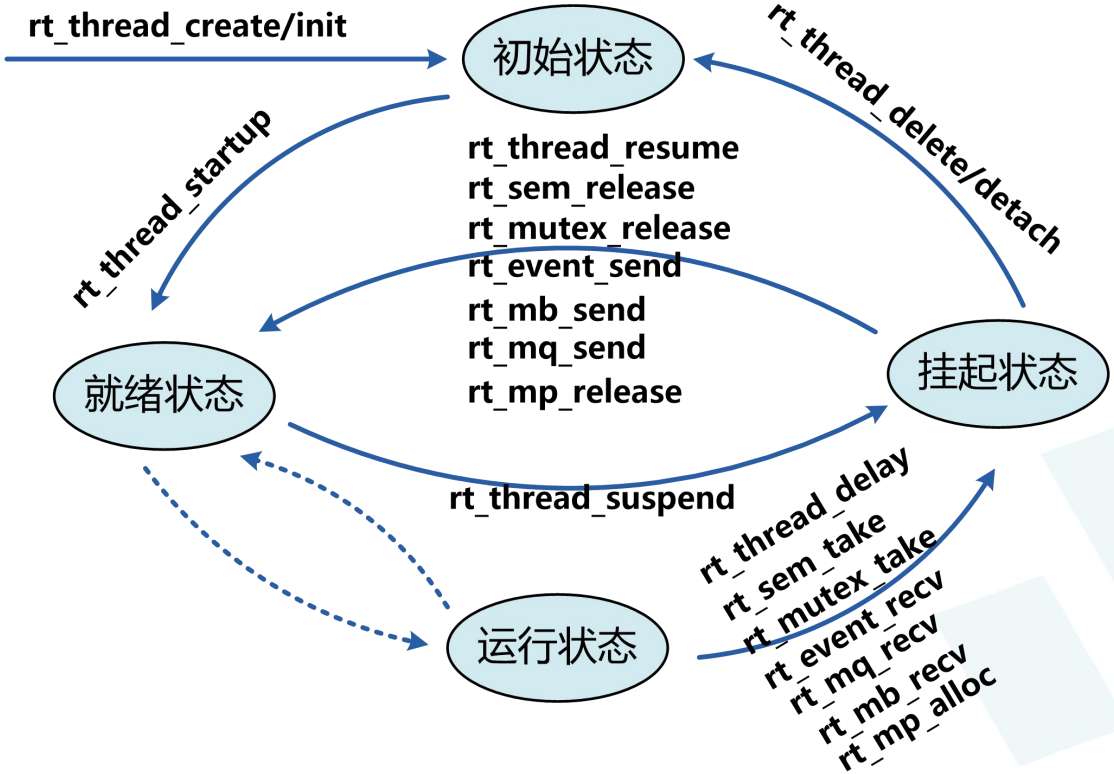
- 我执行一个任务，比如使用饮水机接水。我拿了一个杯子接水，而我必须在饮水机前面等着水接完，这种就是阻塞式线程。

- 如果我拿了杯子接水，把杯子放到饮水机下面，智能饮水机会在杯子接满水之后，自动对我发出异步通知(比如声音告警)。我可以在此期间做其他的事情，这种就是非阻塞式线程（Non-Blocking）

RT-Thread

典型概念——挂起

内核入门——典型概念

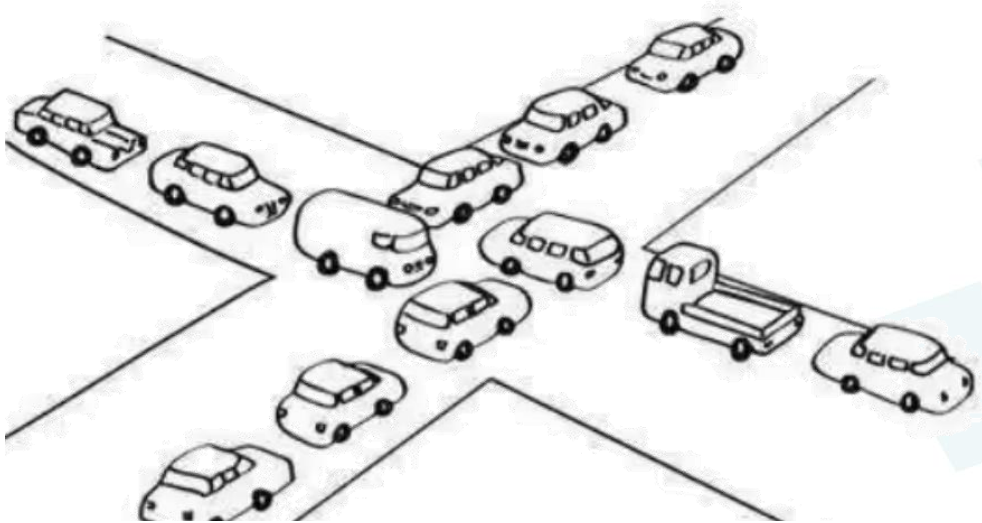

rt_thread_create/init
初始状态
rt_thread_delete/detach
rt_thread_startup
rt_thread_resume
rt_sem_release
rt_mutex_release
rt_event_send
rt_mb_send
rt_mq_send
rt_mp_release
就绪状态
挂起状态
rt_thread_suspend
rt_thread_delay
rt_sem_take
rt_mutex_take
rt_event_recv
rt_mq_recv
rt_mb_recv
rt_mp_alloc
运行状态

RT-Thread

典型概念——死锁

内核入门——典型概念

在十字路口A车道的A1车向转向B车道，但B车道入口被B1车占用；在十字路口B车道的B1车想转向C车道，但C车道入口被C1车占用；在十字路口C车道的C1车想转向D车道，但D车道入口被D1车占用；在十字路口D车道的D1车想转向A车道，但A车道入口被A1车占用；也就是说：线程因为资源竞争，彼此需要资源又都无法释放，导致线程无法获取下一步执行所需的资源，导致死锁产生。



RT-Thread

内核入门——信号量

RT-Thread

信号量——示例

内核入门——信号量

RT-Thread

- 信号量(Semaphore)是一种轻型的用于解决线程间同步问题的内核对象，一个或多个运行线程可以获取或释放它，从而达到同步或互斥的目的。用于实现任务与任务之间、任务与中断处理程序之间的同步与互斥。

- 信号量一般分为三种：

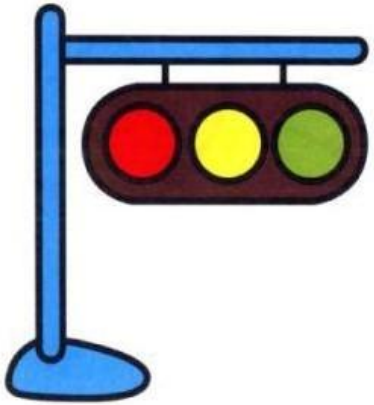
  ~~✓ 互斥信号量~~ 用于解决互斥问题。它比较特殊，可能会引起优先级反转问题

  ✓ 二值信号量 用于解决同步问题

  ✓ 计数信号量 用于解决资源计数问题

- 将信号量进行种类细分，可以根据其用途，在具体实现时做专门处理，提高执行效率和可靠性。

内核入门——信号量



“红灯停、绿灯行”，这是生活中的交通信号，它为司机与行人提供一种信号：

看到红色的灯：停下来等待

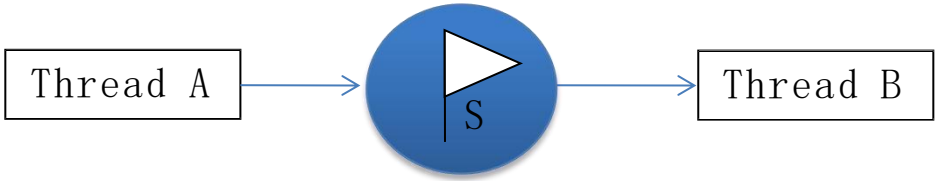
看到绿色的灯：行驶

- 生活中的交通信号灯扮演了操作系统内信号量的角色。
- 根据这种设定好的规则，人们获取到交通系统所发出的不同信号时，就会根据信号的不同而做出不同的举动，那么这就是“同步”的概念。
- 一般在裸机中，会设置一些全局变量flag，代码根据flag的变化，做出对应的动作。

内核入门——信号量

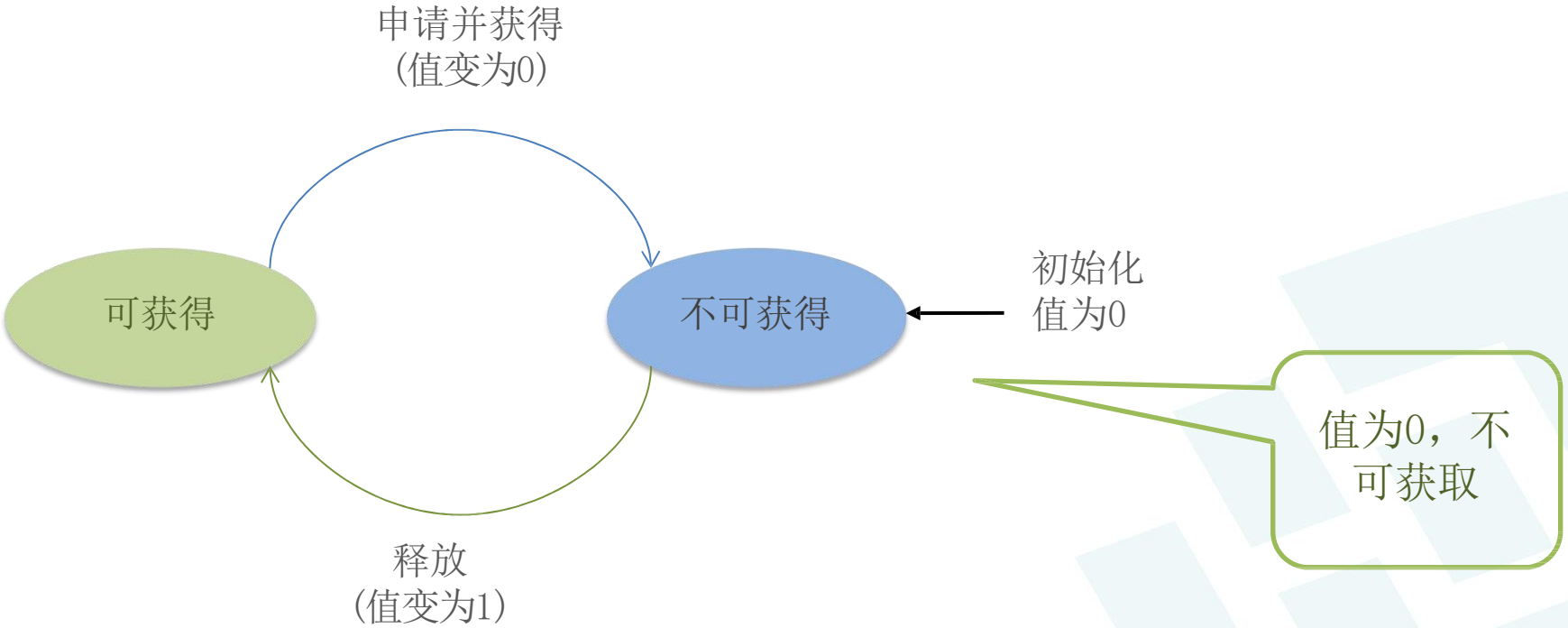
二值信号量主要用于线程与线程之间、线程与中断服务程序(ISR)之间的同步。

- 用于同步的二值信号量初始值为0，表示同步事件尚未产生；
- 线程获取信号量以等待该同步事件的发生；
- 另一个任务或 ISR 到达同步点时，释放信号量（将其值设置为1）表示同步事件已发生，唤醒等待的任务。

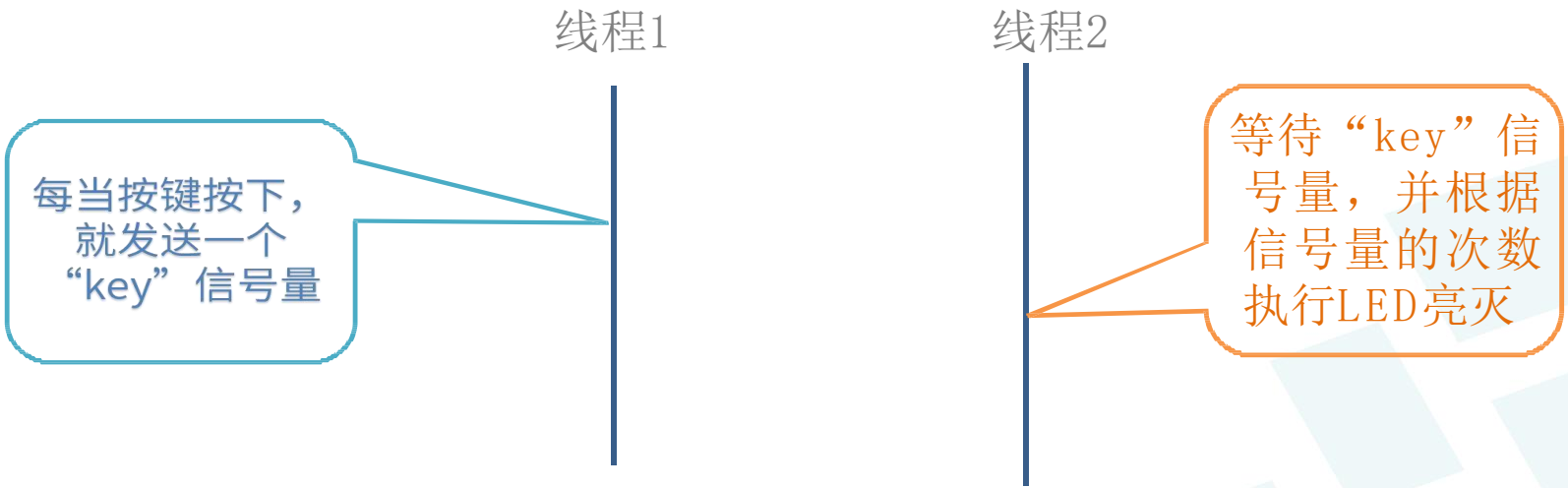


二值信号量
初值为0

内核入门——信号量

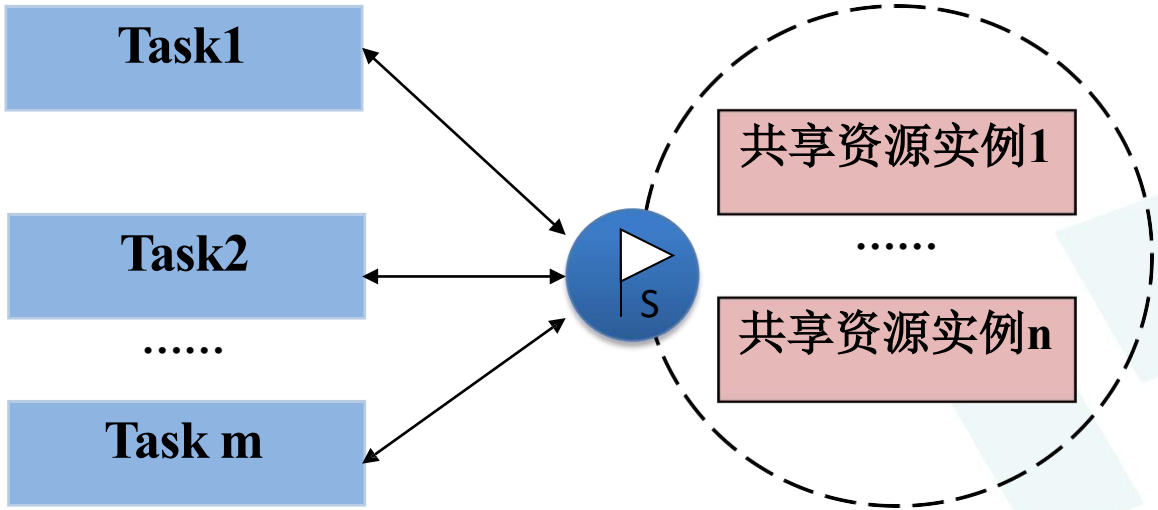


内核入门——信号量

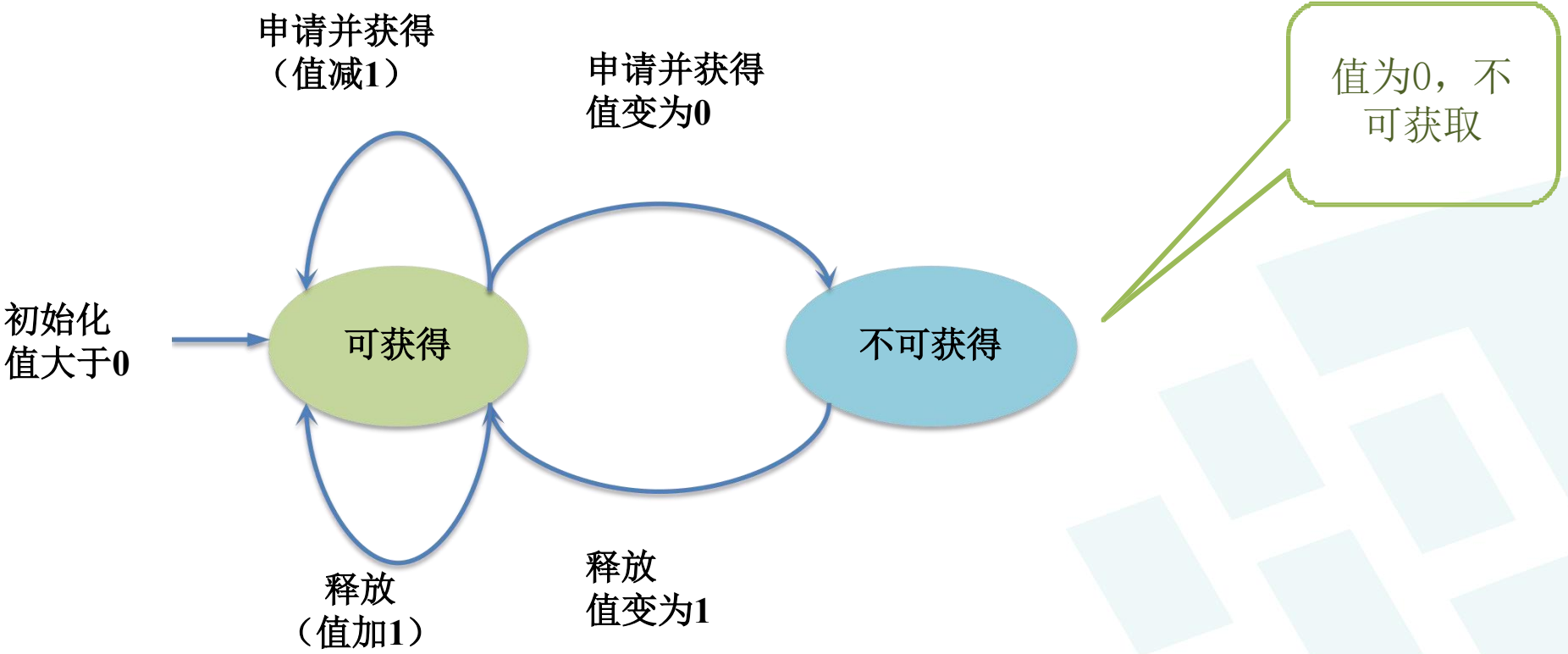


内核入门——信号量

- 计数信号量用于控制系统中共享资源的多个实例的使用，允许多个线程同时访问同一种资源的多个实例。
- 计数信号量被初始化为n（非负整数），n为该种共享资源的数目。



内核入门——信号量



计数信号量状态图

RT-Thread

信号量——API

内核入门——信号量

RT-Thread

- 创建/初始化信号量
- 删除/脱离信号量
- 获取/释放信号量
- 尝试获取信号量

内核入门——创建信号量

RT-Thread

```
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。

内核入门——删除信号量

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

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。

当调用这个函数时，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。

内核入门——初始化信号量

```
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。

内核入门——脱离信号量

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

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。

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。

内核入门——获取信号量

RT-Thread

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

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。

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。

内核入门——无等待获取信号量

RT-Thread

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

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。

这个函数与 rt_sem_take(sem, RT_WAITING_NO) 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。

内核入门——释放信号量

RT-Thread

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

释放信号量可以唤醒挂起在该信号量上的线程

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。

RT-Thread

内核入门——互斥量

RT-Thread

互斥量——示例

内核入门——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。它和信号量不同的是，它支持：

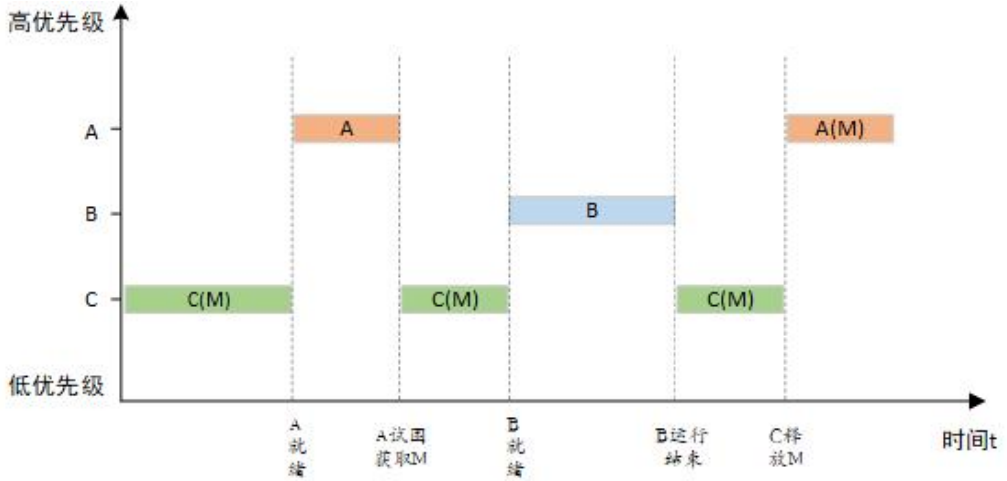
1. 互斥量所有权；
2. 递归访问；
3. 防止优先级翻转的特性。

优先级反转(Priority Inversion)：

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。

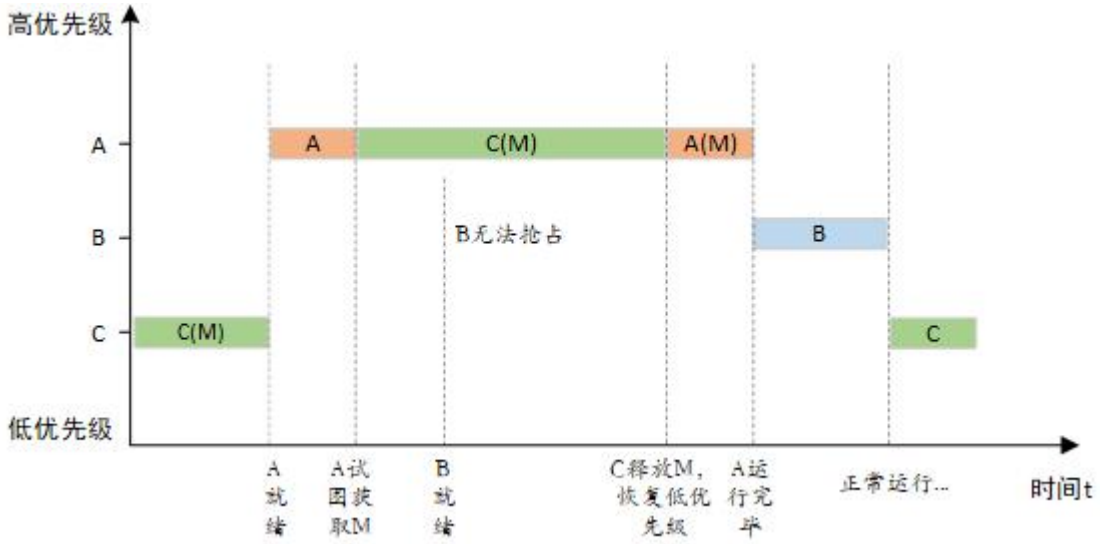
内核入门——互斥量

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，且线程B没有用到共享资源 M ，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。



内核入门——互斥量

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协议（Sha，1990）。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。



RT-Thread

互斥量——API

内核入门——互斥量

- 创建/初始化互斥量
- 删除/脱离互斥量
- 获取/释放互斥量
- 尝试获取互斥量（无等待）

内核入门——创建互斥量

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

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。

内核入门——删除互斥量

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

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。

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。

内核入门——初始化互斥量

RT-Thread

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

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。

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。

内核入门——脱离互斥量

RT-Thread

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

当使用该函数接口后，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。

内核入门——获取互斥量

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

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。

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。

内核入门——无等待获取互斥量

RT-Thread

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

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。

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。

内核入门——释放取互斥量

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

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。

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，等待在该互斥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。

RT-Thread

内核入门——事件集

事件集——示例

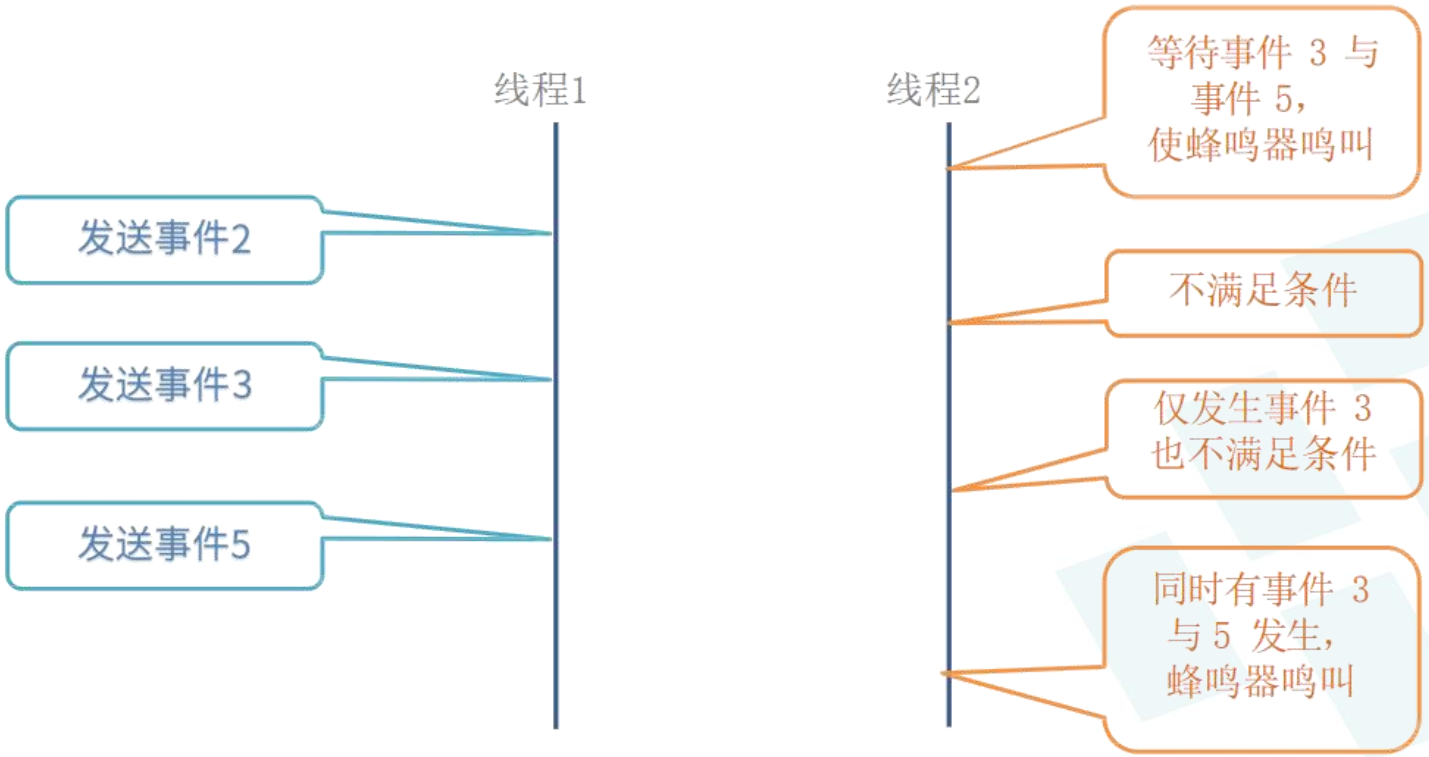
内核入门——事件集

事件集是一个32 bit 的数，每个事件用一个bit位代表；

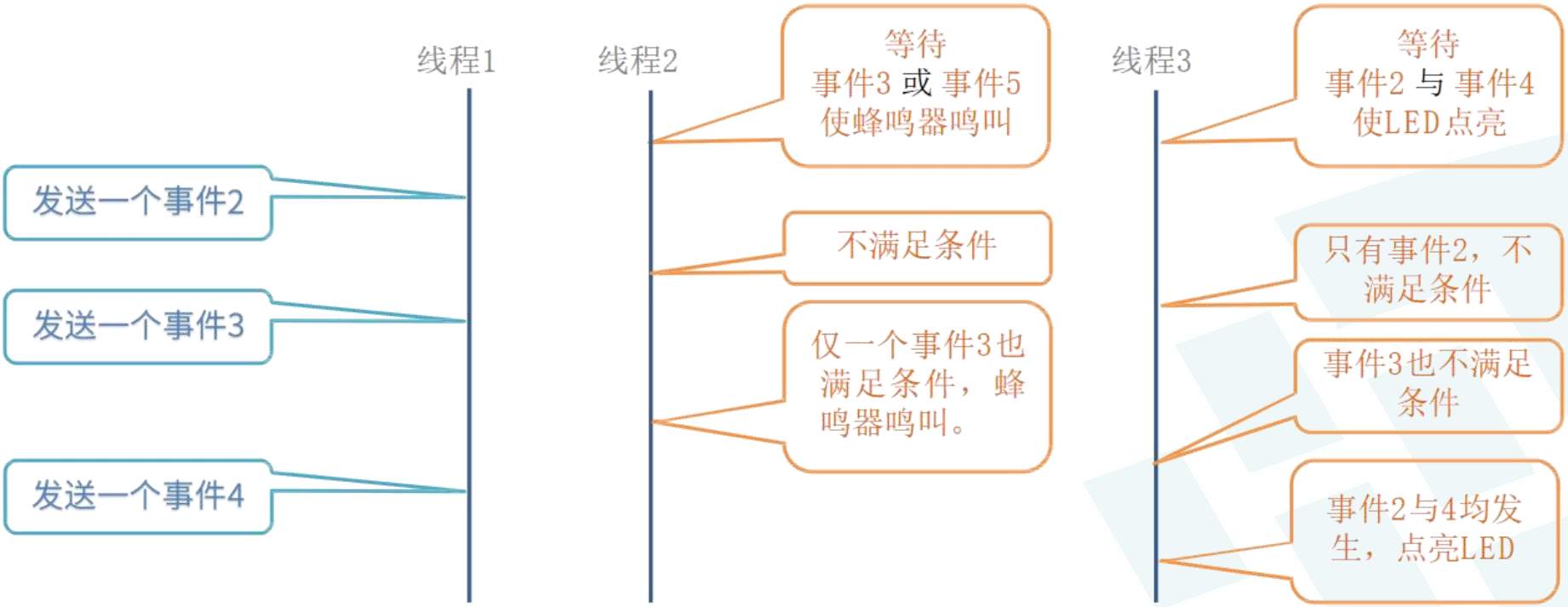
触发方式有与触发、或触发

- ✓ 发送：可以从中断或者线程中进行发送
- ✓ 接收：线程接收，条件检查（逻辑与方式、逻辑或方式）

内核入门——事件集


线程1
发送事件2
发送事件3
发送事件5
线程2
等待事件 3 与
事件 5，
使蜂鸣器鸣叫
不满足条件
仅发生事件 3
也不满足条件
同时有事件 3
与 5 发生，
蜂鸣器鸣叫

内核入门——事件集


线程1
发送一个事件2
发送一个事件3
发送一个事件4
线程2
等待
事件3 或 事件5
使蜂鸣器鸣叫
不满足条件
仅一个事件3也满足条件，蜂鸣器鸣叫。
线程3
等待
事件2 与 事件4
使LED点亮
只有事件2，不满足条件
事件3也不满足条件
事件2与4均发生，点亮LED

RT-Thread

事件集——API

内核入门——事件集

- 创建/初始化事件集
- 删除/脱离事件集
- 获取/释放事件集

内核入门——创建事件

```
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。

内核入门——删除事件

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

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。

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。

内核入门——初始化事件

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

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。

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。

内核入门——脱离事件

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

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。

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

内核入门——发送事件

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

发送事件函数可以发送事件集中的一个或多个事件。

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。

内核入门——接收事件

RT-Thread

```
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);
```

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。

内核入门——接收事件

RT-Thread

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。

RT-Thread

内核入门——消息邮箱

消息邮箱——示例

内核入门——消息邮箱

RT-Thread 操作系统的邮箱用于线程间通信，特点是开销比较低，效率较高。邮箱中的每一封邮件只能容纳固定的 4 字节内容（针对 32 位处理系统，指针的大小即为 4 个字节，所以一封邮件恰好能够容纳一个指针）。典型的邮箱也称作交换消息，如下图所示，线程或中断服务例程把一封 4 字节长度的邮件发送到邮箱中，而一个或多个线程可以从邮箱中接收这些邮件并进行处理。

消息邮箱——API

内核入门——消息邮箱

- 创建/初始化消息邮箱
- 删除/脱离消息邮箱
- 发送/接收消息邮箱

内核入门——创建邮箱

```
rt_mailbox_t rt_mb_create (const char* name, rt_size_t size, rt_uint8_t flag);
```

创建邮箱对象时会先从对象管理器中分配一个邮箱对象，然后给邮箱动态分配一块内存空间用来存放邮件，这块内存的大小等于邮件大小（4 字节）与邮箱容量的乘积，接着初始化接收邮件数目和发送邮件在邮箱中的偏移量。

内核入门——删除邮箱

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

当用 rt_mb_create() 创建的邮箱不再被使用时，应该删除它来释放相应的系统资源，一旦操作完成，邮箱将被永久性的删除。

删除邮箱时，如果有线程被挂起在该邮箱对象上，内核先唤醒挂起在该邮箱上的所有线程（线程返回值是 -RT_ERROR），然后再释放邮箱使用的内存，最后删除邮箱对象。

内核入门——初始化邮箱

RT-Thread

```
rt_err_t rt_mb_init(rt_mailbox_t mb,
                    const char* name,
                    void* msgpool,
                    rt_size_t size,
                    rt_uint8_t flag)
```

初始化邮箱跟创建邮箱类似，只是初始化邮箱用于静态邮箱对象的初始化。与创建邮箱不同的是，静态邮箱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，其余的初始化工作与创建邮箱时相同。

初始化邮箱时，该函数接口需要获得用户已经申请获得的邮箱对象控制块，缓冲区的指针，以及邮箱名称和邮箱容量（能够存储的邮件数）。

内核入门——脱离邮箱

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

脱离邮箱将把静态初始化的邮箱对象从内核对象管理器中脱离。

使用该函数接口后，内核先唤醒所有挂在该邮箱上的线程（线程获得返回值是 - RT_ERROR），然后将该邮箱对象从内核对象管理器中脱离。

内核入门——发送邮件

```
rt_err_t rt_mb_send (rt_mailbox_t mb, rt_uint32_t value);
```

线程或者中断服务程序可以通过邮箱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，一个整型值或者一个指向缓冲区的指针。当邮箱中的邮件已经满时，发送邮件的线程或者中断程序会收到 -RT_EFULL 的返回值。

内核入门——等待方式发送邮件

RT-Thread

```
rt_err_t rt_mb_send_wait (rt_mailbox_t mb,
                          rt_uint32_t value,
                          rt_int32_t timeout);
```

rt_mb_send_wait() 与 rt_mb_send() 的区别在于有等待时间，如果邮箱已经满了，那么发送线程将根据设定的 timeout 参数等待邮箱中因为收取邮件而空出空间。如果设置的超时时间到达依然没有空出空间，这时发送线程将被唤醒并返回错误码。

内核入门——发送紧急邮件

RT-Thread

```
rt_err_t rt_mb_urgent (rt_mailbox_t mb, rt_ubase_t value);
```

发送紧急邮件的过程与发送邮件几乎一样，唯一的不同是，当发送紧急邮件时，邮件被直接插队放入了邮件队首，这样，接收者就能够优先接收到紧急邮件，从而及时进行处理。

内核入门——接收邮件

```
rt_err_t rt_mb_recv (rt_mailbox_t mb, rt_uint32_t* value, rt_int32_t timeout);
```

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并返回 RT_EOK 的返回值，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。

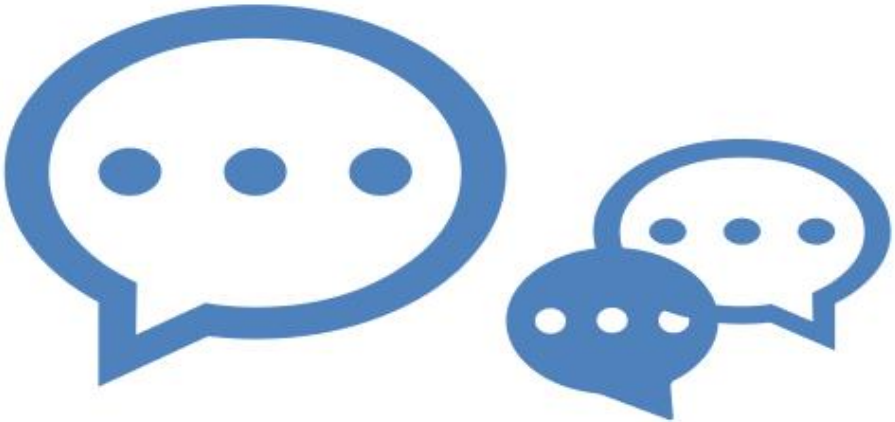
接收邮件时，接收者需指定接收邮件的邮箱句柄，并指定接收到的邮件存放位置以及最多能够等待的超时时间。如果接收时设定了超时，当指定的时间内依然未收到邮件时，将返回 - RT_ETIMEOUT。

RT-Thread

内核入门——消息队列

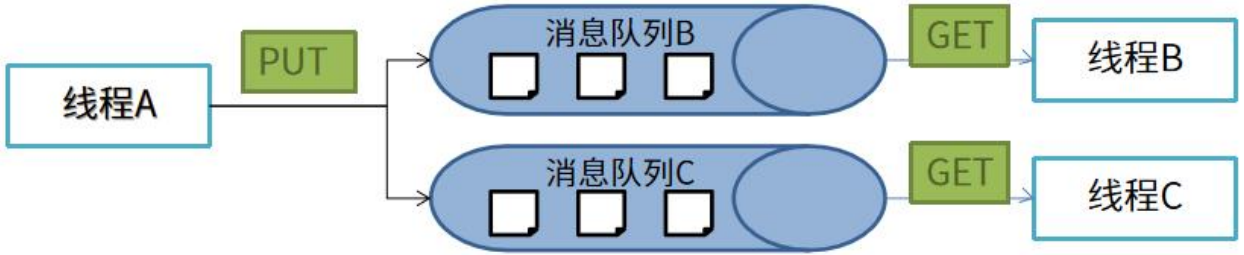
3. 内核入门——消息队列

在实际应用中我们常常遇到一些任务需要和其他任务之间进行数据交流，其实就是不同任务之间的消息传递。在一般的裸机系统中可以通过全局变量来实现不同应用程序之间的数据交互和消息传递。在操作系统中为了更好的管理不同任务之间的消息传递我们引入消息队列的概念。



4. 内核入门——消息队列的概念

消息队列，也就是将多条消息排成的队列形式，是一种常用的线程间通信方式，可以应用在多种场合，线程间的消息交换，使用串口接收不定长数据等。线程可以将一条或多条消息放到消息队列中，同样一个或多个线程可以从消息队列中获得消息；同时消息队列提供异步处理机制可以起到缓冲消息的作用。



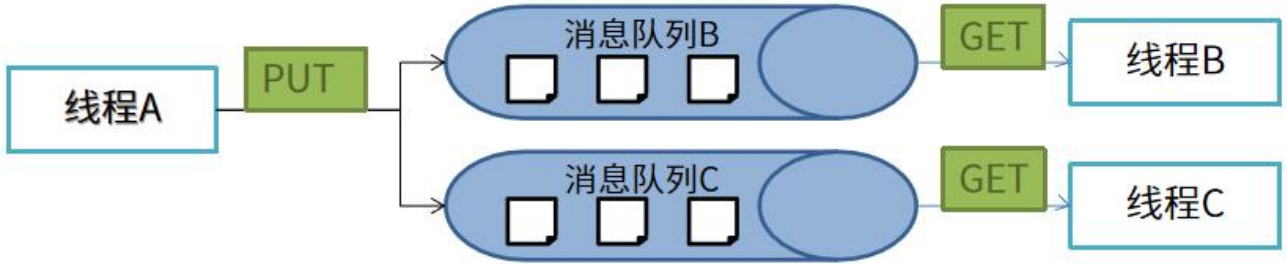
4. 内核入门——消息队列特性

使用消息队列实现线程间的异步通信工作，具有以下特性：

- ✓ 支持读消息超时机制
- ✓ 支持等待方式发送消息
- ✓ 允许不同长度（不超过队列节点最大值）任意类型消息
- ✓ 支持发送紧急消息

4. 内核入门——消息队列工作原理

线程或中断服务例程可以将一条或多条消息放入消息队列中。同样，一个或多个线程也可以从消息队列中获得消息。当有多个消息发送到消息队列时，通常将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）。



4. 内核入门——消息队列的创建

RT-Thread

创建消息队列时先从对象管理器中分配一个消息队列对象，然后给消息队列对象分配一块内存空间，组织成空闲消息链表，这块内存的大小 = [消息大小 + 消息头(用于链表连接)的大小] * 消息队列最大个数，接着再初始化消息队列；接口返回RT_EOK表示动态消息队列创建成功。

```
rt_mq_t rt_mq_create(const char* name,     // 消息队列名称
                     rt_size_t msg_size,    // 消息队列中一条消息的最大长度，单位字节
                     rt_size_t max_msgs,   // 消息队列的最大个数
                     rt_uint8_t flag);       // 消息队列采用的等待方式
```

参数 flag 可以取如下数值： RT_IPC_FLAG_FIFO(先进先出) 或 RT_IPC_FLAG_PRIO(优先级等待)

4. 内核入门——消息队列参数flag

在创建消息队列指定的参数中，事件集标志参数决定了当消息不可获取时，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消息；

- ✓ 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消息。

RT-Thread

✓ 动态：

```
/* 消息队列句柄 */
rt_ert_mq_t mq = RT_NULL;

/* 创建消息队列对象，返回消息队列句柄 */
mq = rt_mq_create("mq", 4, 10, RT_IPC_FLAG_FIFO);  // 名称，消息长度，消息个数，等待方式
```

4. 内核入门——API：发送消息

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，消息队列对象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。当且仅当空闲消息链表上有可用的空闲消息块时，发送者才能成功发送消息；当空闲消息链表上无可用消息块，说明消息队列已满，此时，发送消息的的线程或者中断程序会收到一个错误码（-RT_EFULL）。发送消息的函数接口如下：

```
rt_err_t rt_mq_send (rt_mq_t mq,     // 消息队列对象的句柄
                     void* buffer,   // 消息内容
                     rt_size_t size); // 消息大小
```

4. 内核入门——API：等待方式发送消息

RT-Thread

等待发送消息的函数接口如下，rt_mq_send_wait() 与 rt_mq_send() 的区别在于有等待时间，如果消息队列已经满了，那么发送线程将根据设定的 timeout 参数进行等待。如果设置的超时时间到达依然没有空出空间，这时发送线程将被唤醒并返回错误码。

```
rt_err_t rt_mq_send_wait(rt_mq_t    mq,          // 消息队列对象的句柄
                         const void *buffer,     // 消息内容
                         rt_size_t  size,        // 消息大小
                         rt_int32_t timeout);    // 超时时间
```

4. 内核入门——API：发送紧急消息

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，接收者就能够优先接收到紧急消息，从而及时进行消息处理。发送紧急消息的函数接口如下：

```
rt_err_t rt_mq_urgent(rt_mq_t mq,        // 消息队列对象的句柄
                      void* buffer,      // 消息内容
                      rt_size_t size);   // 消息大小
```

4. 内核入门——API：接收消息

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。接收消息函数接口如下

```
rt_err_t rt_mq_recv (rt_mq_t mq,        // 消息队列对象的句柄
                     void* buffer,      // 消息内容
                     rt_size_t size,    // 消息大小
                     rt_int32_t timeout); // 指定的超时时间
```

接收消息时，接收者需指定存储消息的消息队列对象句柄，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里。此外，还需指定未能及时取到消息时的超时时间，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。

RT-Thread

课后作业

RT-Thread

谢谢！