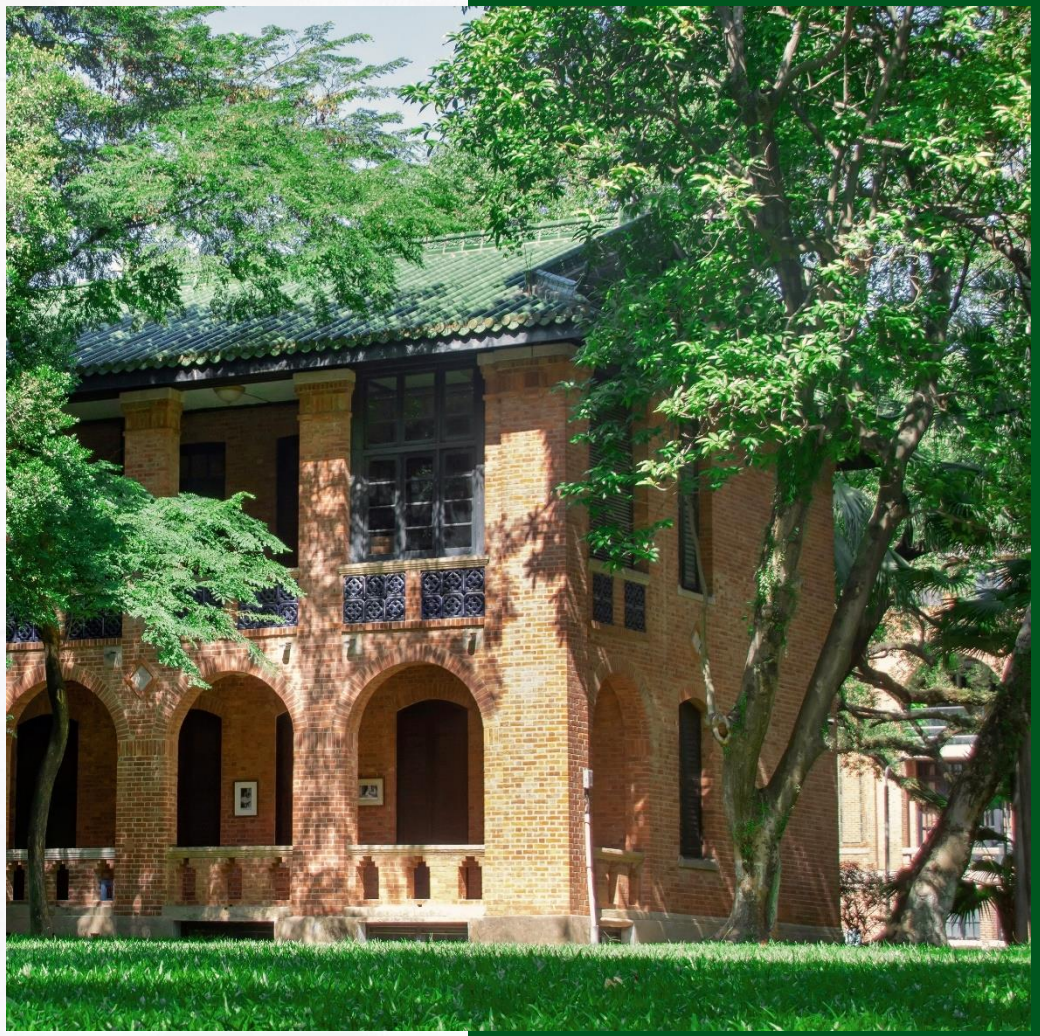
中山大學
SUN YAT-SEN UNIVERSITY

鸿蒙操作系统
并发控制



HarmonyOS中的并发控制

1. **进程间通信（IPC）**专门指操作系统提供的供**进程之间**共享数据的机制。
2. IPC机制在内核设计过程中起着非常重要的作用，是操作系统**内核效率**的决定性因素。
3. 不合理的使用IPC通信会对应用性能造成影响。
   - 在应用主线程中进行IPC通信，消息的发送和接收**需要等待对方进程的响应**，这会**对应用主线程造成阻塞**。如果操作耗时较长或者频率较高，产生的时延会引起页面卡顿、丢帧。
   - 进行IPC通信时需要**进程上下文切换**，从一个进程或线程切换到另一个进程或线程，**会造成CPU时间片的浪费**，从而降低应用性能。

- **HarmonyOS中的并发控制**

4. HarmonyOS支持的IPC手段

  - **事件（event）**
  - **互斥锁（mutex）**
  - **消息队列（queue）**
  - **信号量（semaphore）**

事件运作原理

事件（Event）：一种任务间的通信机制，可用于任务间的同步操作。（当给定的一个或多个事件发生时，一个或者多个任务可以执行，否则任务等待/阻塞）

1. 事件的特点

- 任务间的事件同步，可以一对多，也可以多对多。一对多表示一个任务可以等待多个事件，多对多表示多个任务可以等待多个事件。但是一次写事件最多触发一个任务从阻塞中醒来。
- 多次向事件控制块写入同一事件类型，在被清零前等效于只写入一次。
- 只做任务间同步，不传输具体数据。

事件运作原理

事件（Event）：一种任务间的通信机制，可用于多任务间的同步操作。（当给定的一个或多个事件发生时，一个或者多个任务可以执行，否则任务等）

2. HarmonyOS提供了事件的**初始化**、**读写**、**清零和销毁**等接口。

3. 事件控制块包含的信息

   - UINT32 uwEventID：事件集合，用于标识该任务发生的事件类型，其中每一位表示一种事件类型（0表示该事件类型未发生、1表示该事件类型已经发生），一共31种事件类型，第25位系统保留。
   - LOS_DL_LIST stEventList：等待特定事件的任务链表。

Task3的执行需要Task1和Task2产生相应的事件，才能执行。
事件仅仅标识是否完成，是否任务执行所等待的条件满足了。

事件运作原理

1. **事件初始化：**创建一个事件控制块，该控制块维护一个已处理的事件集合，以及等待特定事件的任务链表。

2. **事件写：**会向事件控制块写入指定的事件，事件控制块更新事件集合，并遍历任务链表，根据任务等待具体条件满足情况决定是否唤醒相关任务。

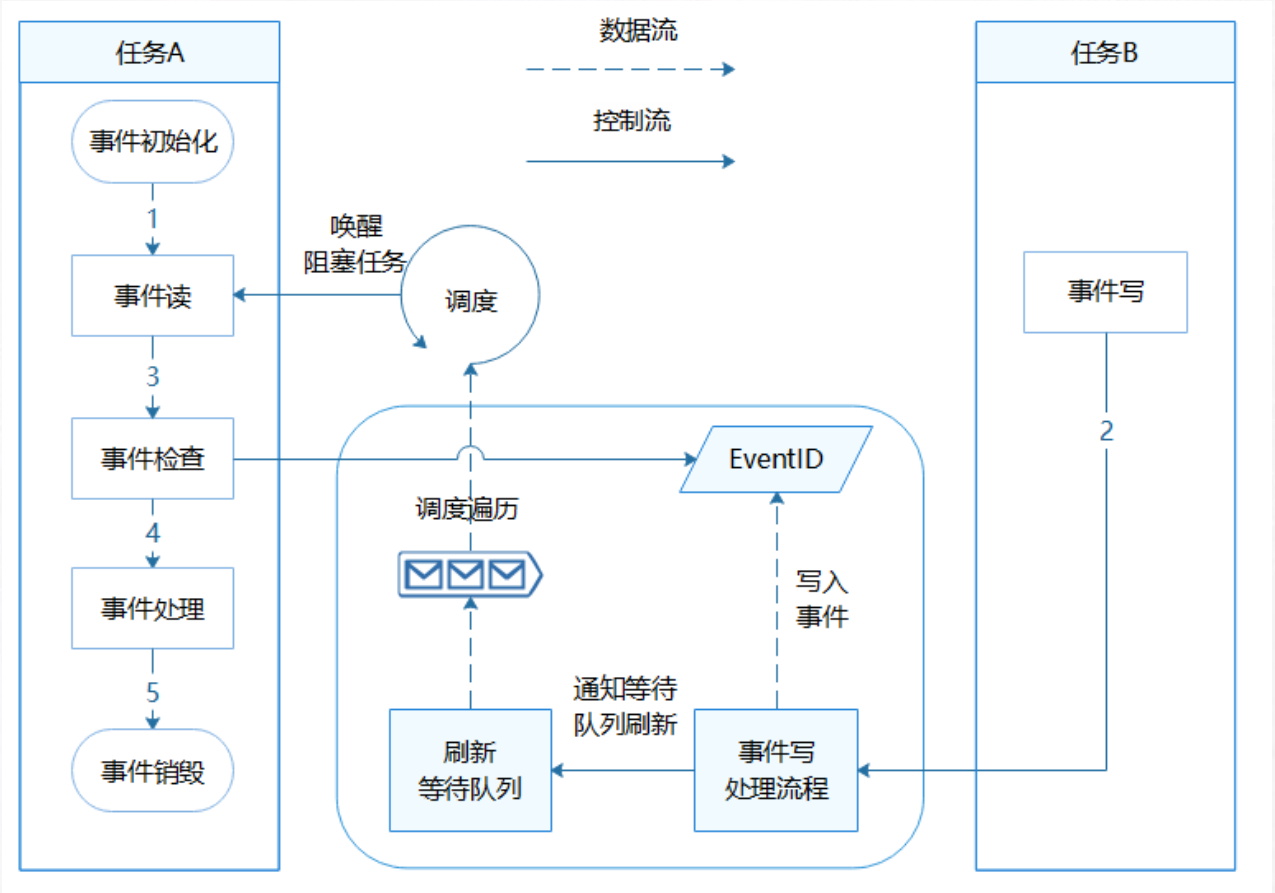


图5-1 事件的运作原理

1. 事件发生可以立即唤醒等待中的任务，而不需要用户轮询查询（对比全局数组）
2. 事件的读写一致性问题（对比全局数组）
3. 超时功能

- **事件运作原理**

3. **事件读：**如果读取的事件已存在时，会直接同步返回。其他情况会根据超时时间以及事件触发情况，来决定返回时机：等待的事件条件在超时时间耗尽之前到达，阻塞任务会被直接唤醒，否则超时时间耗尽该任务才会被唤醒。

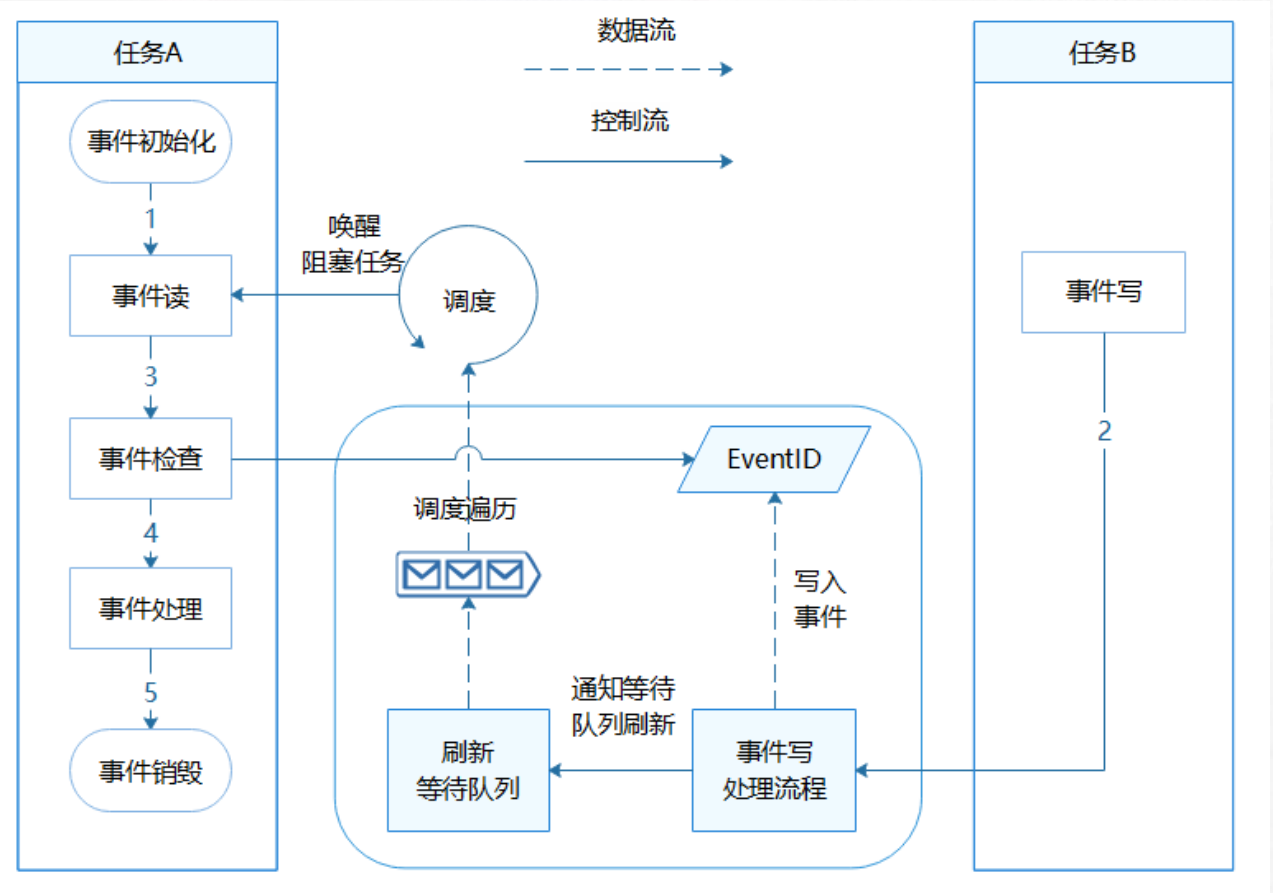


图5-1 事件的运作原理

- **事件运作原理**

4. **事件清零：**根据指定掩码，去对事件控制块的事件集合进行清零操作。当掩码为0时，表示将事件集合全部清零。当掩码为0xffff时，表示不清除任何事件，保持事件集合原状。

5. **事件销毁：**销毁指定的事件控制块。

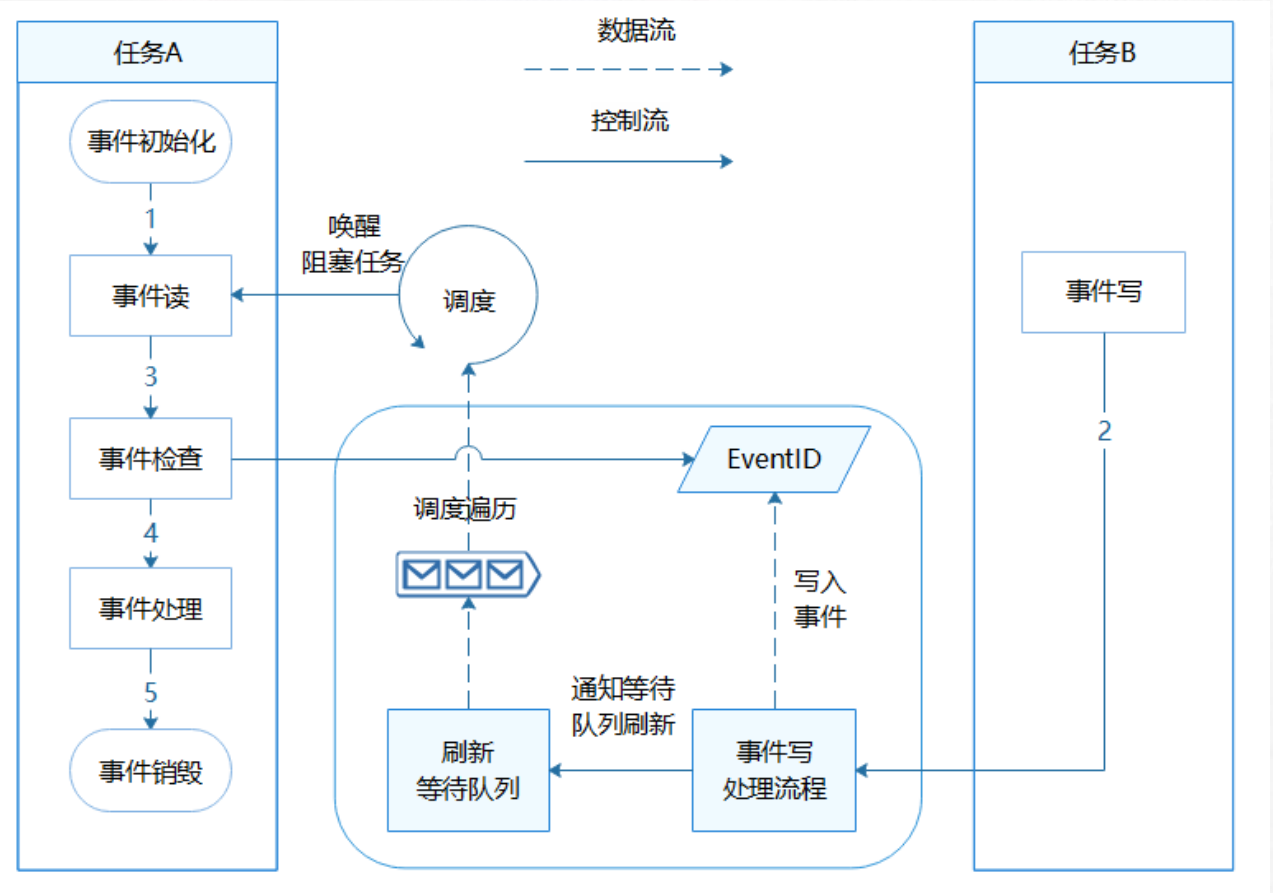


图5-1 事件的运作原理

- **基于事件的任务同步机制**

示例：任务ExampleEvent创建一个任务EventReadTask，EventReadTask读事件阻塞，ExampleEvent向该任务写事件。其中任务EventReadTask优先级高于ExampleEvent。

1. **读进程（EventReadTask）**：调用LOS_EventRead函数来等待写进程就绪。

```
VOID EventReadTask(VOID)
{
    UINT32 ret;
    UINT32 event;

    /* 超时等待方式读事件,超时时间为100 ticks, 若100 ticks后未读取到指定事件，读事件超时，任务直接唤醒 */
    printf("Example_Event wait event 0x%x \n", EVENT_WAIT);

    event = LOS_EventRead(&g_exampleEvent, EVENT_WAIT, LOS_WAITMODE_AND, EVENT_TIMEOUT);
    if (event == EVENT_WAIT) {
        printf("Example_Event, read event :0x%x\n", event);
    } else {
        printf("Example_Event, read event timeout\n");
    }
}
```

- **基于事件的任务同步机制**

2. 写进程（ExampleEvent）：先对事件进行初始化，并在该事件下创建任务。

```
UINT32 ExampleEvent(VOID)
{
    UINT32 ret;
    UINT32 taskId;
    TSK_INIT_PARAM_S taskParam = { 0 };

    /* 事件初始化 */
    ret = LOS_EventInit(&g_exampleEvent);
    if (ret != LOS_OK) {
        printf("init event failed .\n");
        return LOS_NOK;
    }

    /* 创建任务 */
    taskParam.pfnTaskEntry = (TSK_ENTRY_FUNC)EventReadTask;
    taskParam.pcName       = "EventReadTask";
    taskParam.uwStackSize  = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;
    taskParam.usTaskPrio   = 3;
    ret = LOS_TaskCreate(&taskId, &taskParam);
    if (ret != LOS_OK) {
        printf("task create failed.\n");
        return LOS_NOK;
    }
```

- **基于事件的任务同步机制**

2. 写进程（ExampleEvent）： 调用LOS_EventWrite函数执行写入操作。

```
/* 写事件 */
printf("Example_TaskEntry write event.\n");

ret = LOS_EventWrite(&g_exampleEvent, EVENT_WAIT);
if (ret != LOS_OK) {
    printf("event write failed.\n");
    return LOS_NOK;
}
```

2. 写进程（ExampleEvent）：调用LOS_EventClear函数和LOS_EventDestroy对事件进行销毁。

```
/* 清标志位 */
printf("EventMask:%d\n", g_exampleEvent.uwEventID);
LOS_EventClear(&g_exampleEvent, ~g_exampleEvent.uwEventID);
printf("EventMask:%d\n", g_exampleEvent.uwEventID);

/* 删除事件 */
ret = LOS_EventDestroy(&g_exampleEvent);
if (ret != LOS_OK) {
    printf("destory event failed .\n");
    return LOS_NOK;
}
```

基于事件的任务同步机制

osEventFlagsWait函数挂起当前运行线程，直到设置了由参数**ef_id**指定的事件对象中的任何或所有由参数flags指定的事件标志。当这些事件标志被设置，函数立即返回。否则，线程将被置于阻塞状态。

osEventFlagsWait(osEventFlagsld_t ef_id, uint32_t flags, uint32_t options, uint32_t timeout)

读事件（获取事件），在options参数中设置读取模式，来选择用户感兴趣的事件，读取模式如下：

- **所有事件**，LOS_WAITMODE_AND（逻辑与）：读取掩码中所有事件类型，只有读取的所有事件类型都发生了，才能读取成功。
- **任一事件**，LOS_WAITMODE_OR（逻辑或）：读取掩码中任一事件类型，读取的事件中任意一种事件类型发生了，就可以读取成功了。
- **清除事件**，LOS_WAITMODE_CLR：下面两种方法表示事件读取成功后，对应事件类型位会自动清除：

LOS_WAITMODE_AND| LOS_WAITMODE_CLRLOS_WAITMODE_OR| LOS_WAITMODE_CLR

- **基于事件的任务同步机制**

  最终上述程序的**运行结果**:

1. 读进程等待写进程就绪，读进程被阻塞。

3. 读进程被唤醒后继续执行。

```
1  Example_Event wait event 0x1
2  Example_TaskEntry write event.
3  Example_Event, read event :0x1
4  EventMask:1
5  EventMask:0
```

2. 写进程写入事件，此时读进程会被唤醒。因为读进程优先级更高，写进程被抢占。

4和5. 写进程清除事件标志位，标志位由1变为0，表示事件集合清零。

互斥锁

1. 互斥锁又称互斥型信号量，是一种特殊的二值性信号量，用于实现对共享资源的独占式处理。
   - 任意时刻互斥锁的状态只有两种，开锁或闭锁。
   - 当任务持有互斥锁时，该互斥锁处于闭锁状态，这个任务获得该互斥锁的所有权。
   - 当该任务释放互斥锁时，该互斥锁被开锁，任务失去该互斥锁的所有权。
   - 当一个任务持有互斥锁时，其他任务将不能再对该互斥锁进行开锁或持有。
2. 互斥锁属性包含3个属性：协议属性、优先级上限属性和类型属性。协议属性用于处理不同优先级的任务申请互斥锁；类型属性用于标记是否检测死锁，是否支持递归持有。
3. 互斥锁可被用于对共享资源的保护从而实现独占式访问。另外互斥锁可以解决信号量存在的优先级翻转问题。

- **互斥锁的运作原理**

用互斥锁处理非共享资源的同步访问时，如果有任务访问该资源，则互斥锁为加锁状态。此时其他任务如果想访问这个公共资源则会被阻塞，直到互斥锁被持有该锁的任务释放后，其他任务才能重新访问该公共资源。

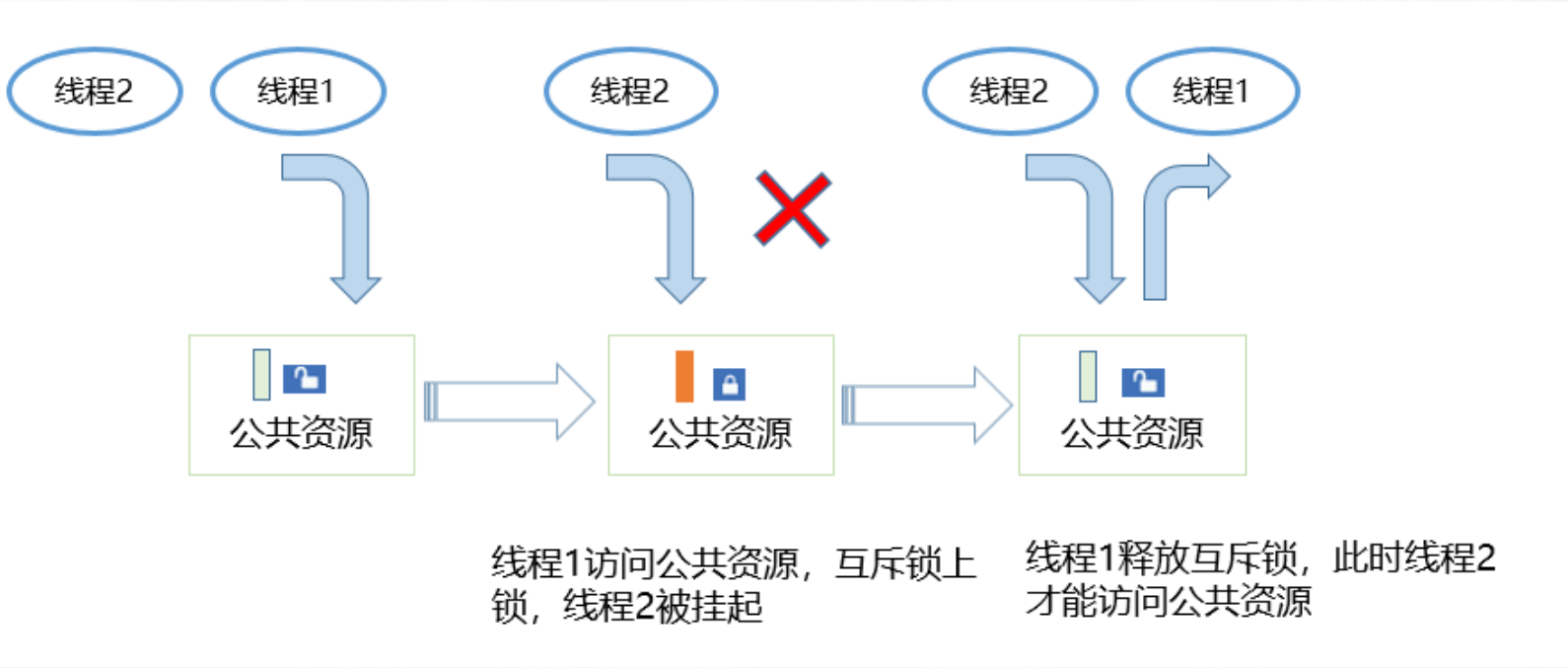


图5-2 互斥锁的运作原理

互斥锁的优先级翻转现象

- 有高、中、低三个优先级的Task，高优先级的Task和低优先级的Task使用了同一个互斥锁。
- 低优先级的Task先一步持有互斥锁。
- 高优先级的Task进入执行时，尝试获取互斥锁，由于其被低优先级的Task占有，因此高优先级的Task被暂时挂起。

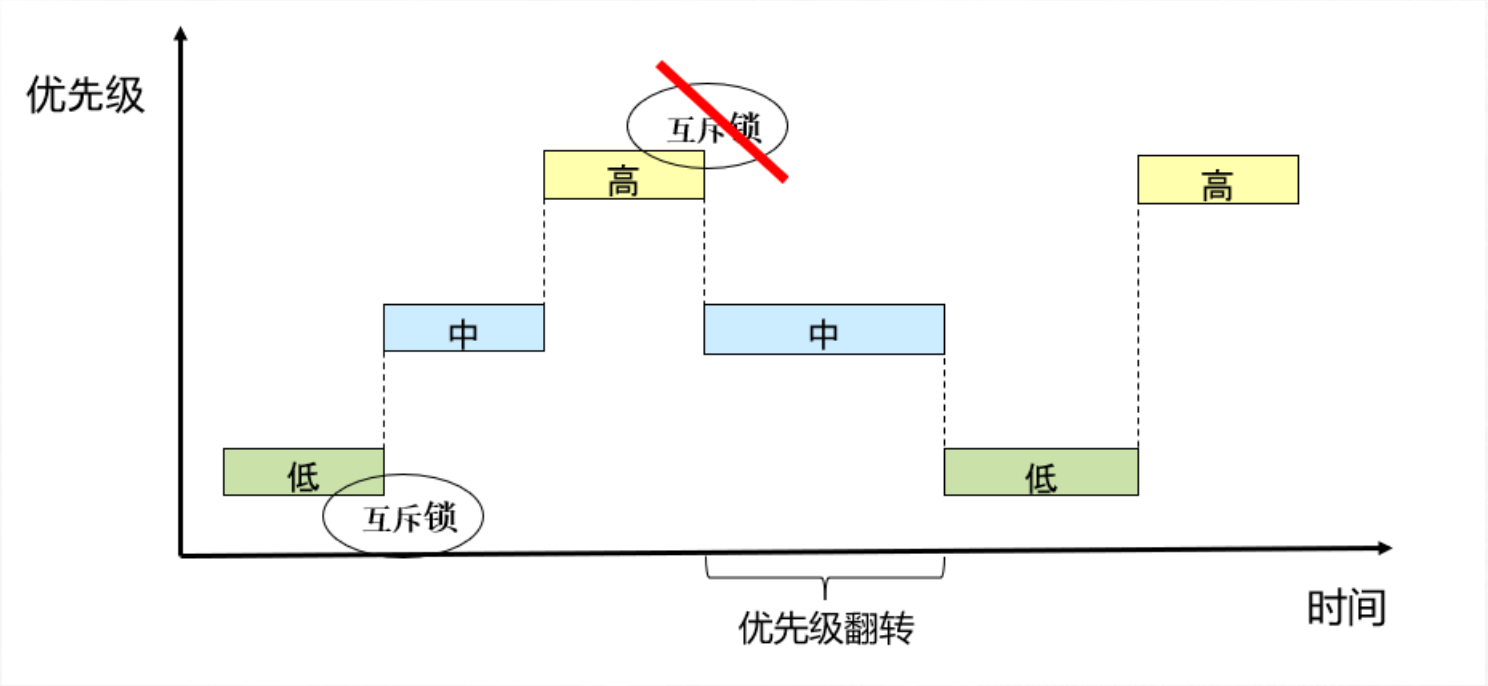


图5-3 互斥锁的优先级翻转现象

- **互斥锁的优先级翻转现象**
  - 由于中优先级Task比低优先级Task权限高，因此中优先级Task继续执行，直到其执行完毕，低优先级的Task才执行。
  - 等到低优先级Task执行完毕，释放互斥锁后，高优先级的Task才能执行。

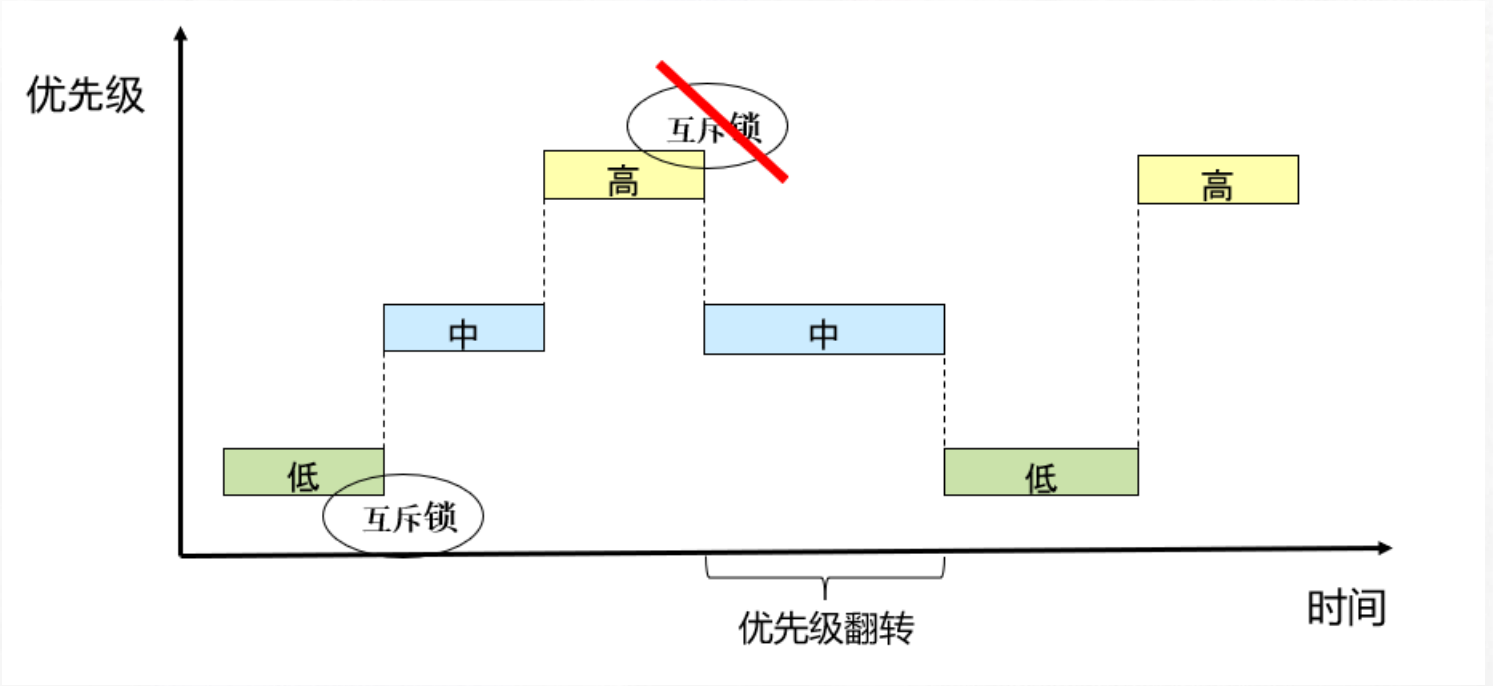


图5-3 互斥锁的优先级翻转现象

互斥锁的优先级翻转现象

为了解决互斥锁的优先级翻转现象，HarmonyOS使用的两种解决方案

1. 优先级继承---互斥锁持有者继承最高优先级的阻塞Task的优先级。
   - 当高优先级Task尝试持有互斥锁但无法获得时，互斥锁所有者会临时被赋予被阻塞Task的优先级。
   - 释放互斥锁时，它将被恢复其原始优先级。
2. 优先级保护---每个互斥锁都会被赋予一个优先级上限。
   - 当Task拥有互斥锁时，如果互斥锁的优先级高于Task自身，则Task会临时接受互斥锁优先级的上限。
   - 释放互斥锁时，它将被恢复其原始优先级。

消息队列

一种常用于任务间通信的数据结构。队列接收来自任务或中断的不固定长度消息，并根据不同的接口确定传递的消息是否存放在队列空间中。

1. 任务能够从队列里面读取消息，当队列中的消息为空时，挂起读取任务；当队列中有新消息时，挂起的读取任务被唤醒并处理新消息。
2. 任务也能够往队列里写入消息，当队列已经写满消息时，挂起写入任务；当队列中有空闲消息节点时，挂起的写入任务被唤醒并写入消息。
3. 可以通过调整读队列和写队列的超时时间来调整读写接口的阻塞模式，如果将读队列和写队列的超时时间设置为0，就不会挂起任务，接口会直接返回，这就是非阻塞模式。反之，如果将读队列和写队列的超时时间设置为大于0的时间，就会以阻塞模式运行。

消息队列

1. 提供了异步处理机制，允许将一个消息放入队列，但不立即处理；同时队列还有缓冲消息的作用，可以使用队列实现任务异步通信。
2. 消息队列的特性：
   - 消息以先进先出的方式排队，支持异步读写。
   - 读队列和写队列都支持超时机制。
   - 每读取一条消息，就会将该消息节点设置为空闲。
   - 发送消息类型由通信双方约定，可以允许不同长度（不超过队列的消息节点大小）的消息。
   - 一个任务能够从任意一个消息队列接收和发送消息。
   - 多个任务能够从同一个消息队列接收和发送消息。
   - 创建队列时所需的队列空间，接口内系统自行动态申请内存。

消息队列的运作原理

1. 创建队列时，创建队列成功会返回队列ID。
2. 在队列控制块中维护着一个消息头节点位置Head和尾节点位置Tail，用于表示当前队列中消息的存储情况。Head表示队列中被占用的消息节点的起始位置。Tail表示被占用的消息节点的结束位置，也是空闲消息节点的起始位置。

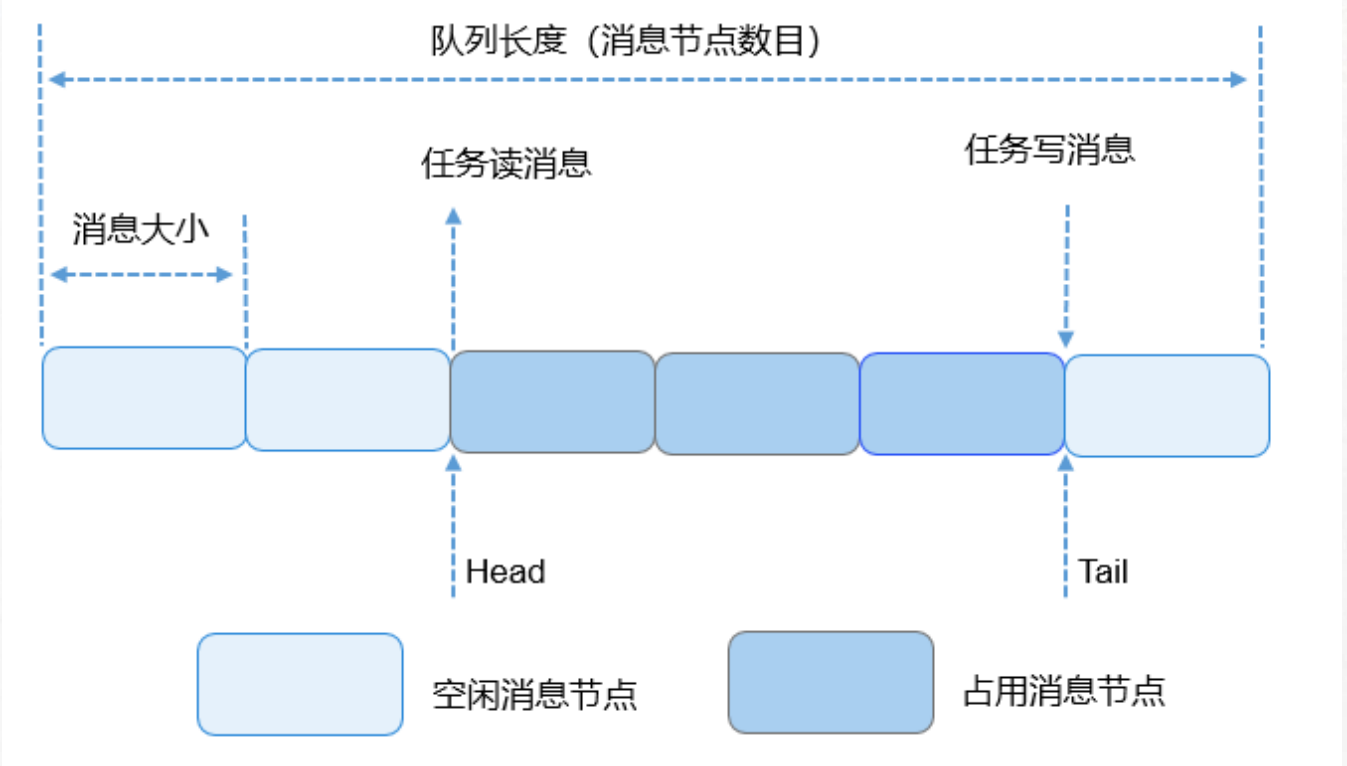


图5-4 消息队列的运作原理

消息队列的运作原理

3. 写队列时，不能对已满队列进行写操作。写队列支持两种写入方式：队列尾节点写入和队列头节点写入。（右图只示意了尾节点写入方式）

4. 读队列时，先判断队列是否有消息需要读取，对全部空闲队列进行读操作会引起任务挂起。如果队列可以读取消息，则根据Head找到最先写入队列的消息节点进行读取。如果Head已经指向队列尾部则采用回卷方式。

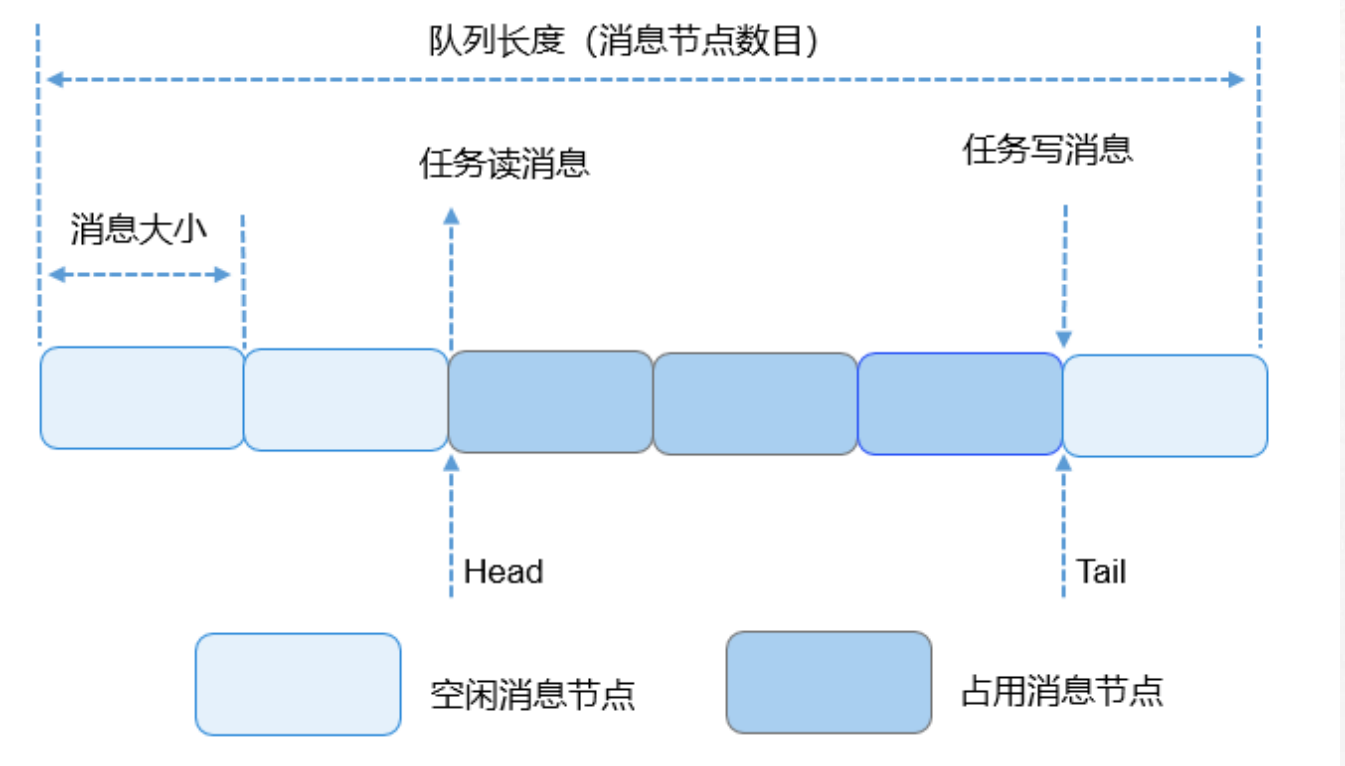


图5-4 消息队列的运作原理

信号量

1. **信号量（Semaphore）**是一种实现任务间通信的机制，可以实现任务间同步或共享资源的互斥访问。

2. 一个信号量的数据结构中，通常有一个计数值，用于对有效资源数的计数，表示剩下的可被使用的共享资源数，其值的含义分两种情况：

   - **=0**：该信号量当前不可获取，因此可能存在正在等待该信号量的任务。
   - **>0**：该信号量当前可被获取。

信号量

3. 以**同步**为目的的信号量和以**互斥**为目的的信号量在使用上有如下不同：

- 用作互斥时，初始信号量计数值不为0，表示可用的共享资源个数。在需要使用共享资源前，先获取信号量，然后使用一个共享资源，使用完毕后释放信号量。这样在共享资源被取完，即信号量计数减至0时，其他需要获取信号量的任务将被阻塞，从而保证了共享资源的互斥访问。而当共享资源数为1时，一般使用二值信号量，它是一种类似于互斥锁的机制。
- 用作同步时，初始信号量计数值为0。任务1因获取不到信号量而阻塞，直到任务2或者某中断释放信号量，任务1才得以进入Ready或Running态，从而达到了任务间的同步。

信号量

信号量允许多个任务在同一时刻访问共享资源，但会限制同一时刻访问此资源的最大任务数目。

- **初始化**：为配置的N个信号量申请内存。
- **创建**：从未使用的信号量链表中获取一个信号量，并设定初值。
- **申请**：若其计数器值大于0，则直接减1返回成功，否则任务阻塞，等待其它任务释放该信号量，等待的超时时间可设定。当任务被一个信号量阻塞时，将该任务挂到信号量等待任务队列的队尾。
- **释放**：若没有任务等待该信号量，则直接将计数器加1返回。否则唤醒该信号量等待任务队列上的第一个任务。
- **删除**：将正在使用的信号量置为未使用信号量，并挂回到未使用链表。

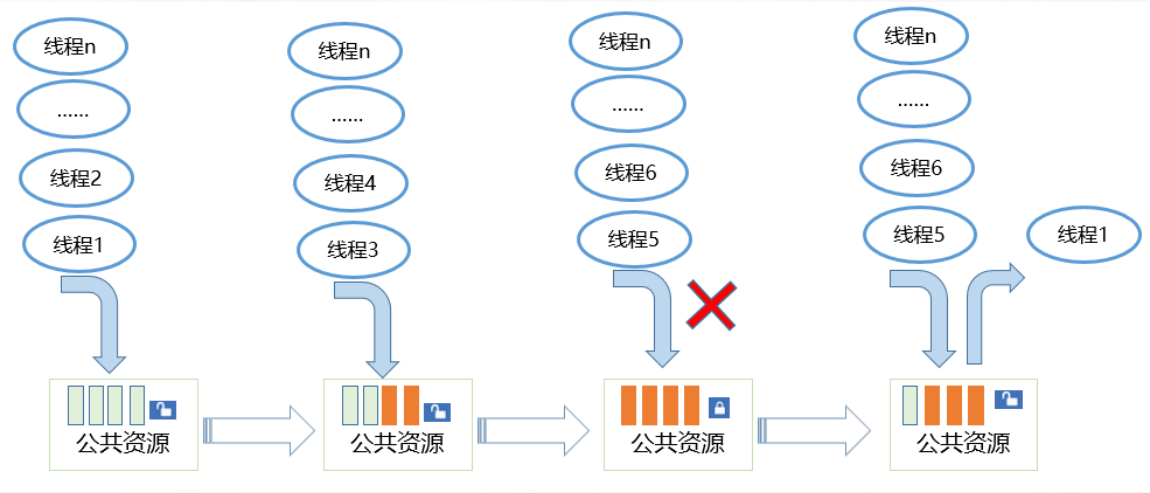


图5-5 信号量的运作原理

中山大學
SUN YAT-SEN UNIVERSITY

谢谢观看

SUN YAT-SEN UNIVERSITY