4.互斥量

2025年10月5日 12:17

- 一、互斥量的引入
  - 在多任务系统中，**多任务可能同时访问同一资源**（比如两个任务都要向串口打印数据），若不加控制会**导致“数据错乱”或“资源异常”**，这一问题被称为**临界资源竞争。**
  - 那么此时就有人说我们之前学习的二值信号量不是也可以实现“互斥”访问吗，确实利用信号量确实可用实现这种对资源的互斥访问，但是信号量存在两个非常明显的缺陷
    - 1.**无“拥有者”属性**：A任务获取信号量之后，B任务可以释放信号量，逻辑上不严谨，易引发误操作。
    - 2.**无法解决优先级反转**：**高优先级的任务被低优先级的任务“间接阻塞”（后文详细说明），导致系统的实时性下降。**
  - 互斥量通过“**谁获取，谁释放**”的规则和“**优先级继承**”机制，完美解决了上述问题，**是FreeRTOS中保护共享资源的“标准方案”。**
- 二、互斥量和普通信号量的关键区别
  - 首先是**拥有者属性**：**互斥量是谁获取了互斥量，必须是谁去释放互斥量，保证了别的任务无权干涉。**而**信号量是任务A即便获取了信号量，也可以有任务B去释放信号量。**
  - **核心用途区别**：互斥量是**保护临界资源，解决资源竞争问题。**而信号量是用于**任务同步（如任务唤醒，事件通知）**
  - **优先级继承机制**：**互斥量支持**，能避免优先级反转。而**信号量不支持**，易出现优先级反转
  - 初始值：互斥量初始值为“可用（1）”，表示资源未被占用。而信号量初始可设置为0（不可用）或1（可用），根据同步需求决定。

二、核心特性：互斥量与普通信号量的关键区别

特性	互斥量（Mutex）	二值信号量（Binary Semaphore）
拥有者属性	有：只有获取互斥量的任务能释放它	无：任意任务可释放已被获取的信号量
核心用途	保护临界资源，解决资源竞争	任务间同步（如“任务唤醒”“事件通知”）
优先级继承	支持：能避免优先级反转	不支持：易出现优先级反转
释放限制	必须由获取任务释放，否则会触发断言（DEBUG 模式）	无限制，任意任务可释放
初始值	初始为“可用”（1），表示资源未被占用	初始可设为 0 或 1，根据同步需求决定

- 三、优先级继承--解决优先级反转的核心
  - 1.优先级反转：实时系统的“隐形杀手”
    - 假设系统中有3个任务，分别是**高优先级任务A（H），中优先级任务B（M），低优先级任务C（L），下文分别用H，M，L表示三个任务及优先级**
    - 1.**开始时L任务获取了信号量**，正在访问串口资源（临界资源）
    - 2.**H任务就绪**，因为其优先级最高，所以要**抢占L任务的CPU使用权**
    - 3.**H任务此时要获取信号量**（也要访问串口），但是此时**信号量已经被L任务占有了**，**H任务只能进入阻塞状态，等待L任务释放信号量**
    - 4.因H任务进入阻塞等待状态，所以此时**L任务重新获得CPU使用权**，但是此时正好**M任务就绪了**，此时**M任务抢占L任务的CPU使用权，导致L任务无法继续执行释放信号量**，因为M任务无需获取信号量，如果M任务不会进入阻塞的话，那么以后将一直执行M任务，H任务和L任务将无法得到执行，如果M任务会进入阻塞，此时L任务继续执行，只有当L任务成功释放信号量之后，H任务才可以得到执行

- 最终结果时：**H任务被M任务间接阻塞，直到M任务执行完，L任务重新执行并释放信号量，H任务才可以执行--这就是“优先级反转”严重破坏了系统的实时性（高优先级任务无法及时响应）**

- 2.优先级继承：互斥量如何修复反转
  - **当L任务获取互斥量，且H任务想要互斥量时，互斥量就会自动触发“优先级继承”**
  - 1.**临时提升L任务的优先级**：**将L任务的优先级提升到与H任务的优先级相同**
  - 2.**避免M任务抢占**：此时**L任务的优先级高于M任务的优先级，M任务无法抢占**，L任务能“快速执行完临界区代码”
  - 3.**释放后恢复优先级**：**L任务释放互斥量后，其优先级自动恢复原始值，H任务立即获取互斥量并执行**
  - 通过这一机制，H任务的阻塞时间被严格限制在L任务执行临界区的时间，**避免了M任务的干扰**，保障实时性

四、互斥量的创建

- 互斥量其本质就是**一种特殊的二值信号量**，所以互斥量**本身就一种信号量**，在FreeRTOS中所有的信号量句柄都要保存在类型为xSemaphoreHandle的变量中
- 这里创建互斥量有一个专门的API函数**xSemaphoreMutex（void）**

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

- 该函数无参数，有返回值，返回一个信号量句柄

```
#if ( ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) && ( configUSE_MUTEXES == 1 ) )
    #define xSemaphoreCreateMutex()    xQueueCreateMutex( queueQUEUE_TYPE_MUTEX )
#endif
```

- 我们可以看到互斥量创建函数是一个宏定义函数，真正的实现函数是右边那个函数，我们可以看到右边的函数默认就会传入一个参数，这个参数也很熟悉，就是前面我们讲信号量创建时最后的一个参数，它决定了创建信号量的种类，很明显这里就是传入的互斥量种类

```
#if ( ( configUSE_MUTEXES == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )

    QueueHandle_t xQueueCreateMutex( const uint8_t ucQueueType )
    {
        QueueHandle_t xNewQueue;
        const UBaseType_t uxMutexLength = ( UBaseType_t ) 1, uxMutexSize = ( UBaseType_t ) 0;

        traceENTER_xQueueCreateMutex( ucQueueType );

        xNewQueue = xQueueGenericCreate( uxMutexLength, uxMutexSize, ucQueueType );
        prvInitialiseMutex( ( Queue_t * ) xNewQueue );

        traceRETURN_xQueueCreateMutex( xNewQueue );

        return xNewQueue;
    }

#endif /* configUSE_MUTEXES */
/*-----------------------------------------------------------*/
```

- 当我深入底层代码我们可以发现，**传入的参数就是后文创建队列时传入的最后一个参数，所以创建互斥量本质还是创建一个队列，这里传入的参数，第一个队列深入，在上面给出了是1，每个队列空间的大小也给出了是0，这两个参数和创建二值信号量的时候的参数一样，但是第三个参数传入的是互斥量种类，这与二值信号量不同。**
- 这是互斥量的创建函数，因为互斥量本质就是一个信号量，**所以互斥量的释放和获取与信号量的一样**，分别是**xSeaphoreGive（）和xSemaphoreTake（）**。
- 注意**互斥量初始时刻就是可以状态**，不和二值信号量一样初始需要先释放一次

五、互斥量的“使用禁忌”

- 互斥量的设计决定了它有严格的限制，违规使用会导致系统崩溃或锁死
- 1.**禁止在中断服务函数（ISR）中使用**
  - 互斥量依赖任务的“阻塞态”和“优先级继承”，而中断不是任务，无法进入阻塞态
- 2.**禁止在任务删除（vTaskDelete（））前不释放互斥量**
  - 若任务获取互斥量被删除后，互斥量会永远处于“被占用”状态，其他任务尝试获取时会进入永久阻塞，导致“锁死”
- 3.**禁止长时间占用互斥量**
  - 互斥量保护的临界区代码应尽可能短（如仅读写全局变量，调试硬件接口），若任务占用互斥量后执行复杂逻辑（如延时，循环），会导致其他等待该互斥量的任务长时间阻塞，影响系统响应速度

六、总结

- 互斥量的三个核心记忆点
- 1.核心作用：**保护临界资源，确保同一时间只有一个任务访问，解决资源竞争**
- 2.核心机制：**通过优先级继承，避免了优先级反转，保障实时系统的响应性**
- 3.核心禁忌：**不许在ISR中使用，不允许获取后不释放，不允许长时间占用**