FreeRTOS 任务调度

前言

FreeRTOS 是一个是实时内核，任务是程序执行的最小单位，也是调度器处理的基本单位，移植了 FreeRTOS ，则避免不了对任务的管理，在多个任务运行的时候，任务切换显得尤为重要。而任务切换的效率会决定了系统的稳定性与效率。

FreeRTOS 的任务切换是干嘛的呢， rtos 的实际是永远运行的是具有最高优先级的运行态任务，而那些之前在就绪态的任务怎么变成运行态使其得以运行呢，这就是我们 FreeRTOS 任务切换要做的事情，它要做的是找到最高优先级的就绪态任务，并且让它获得cpu的使用权，这样，它就能从就绪态变成运行态，这样子，整个系统的实时性就会很好，响应也会很好，而不会让程序阻塞卡死。

要知道怎么实现任务切换，那就要知道任务切换的机制，在不同的 cpu（mcu） 中，触发的方式可能会不一样，现在是以Cortex-M3为例来讲讲任务的切换。

SVC 和 PendSV

SVC（系统服务调用，亦简称系统调用）和 PendSV （ Pended System Call ，可悬起系统调用），它们多用于在操作系统之上的软件开发中。 SVC 用于产生系统函数的调用请求。例如，操作系统不让用户程序直接访问硬件，而是通过提供一些系统服务函数，用户程序使用 SVC 发出对系统服务函数的呼叫请求，以这种方法调用它们来间接访问硬件。因此，当用户程序想要控制特定的硬件时，它就会产生一个 SVC 异常，然后操作系统提供的 SVC 异常服务例程得到执行，它再调用相关的操作系统函数，后者完成用户程序请求的服务。

另一个相关的异常是 PendSV （可悬起的系统调用），它和 SVC 协同使用。一方面， SVC 异常是必须立即得到响应的（若因优先级不比当前正处理的高，或是其它原因使之无法立即响应，将发生硬 fault——译者注），应用程序执行 SVC 时都是希望所需的请求立即得到响应。另一方面，PendSV 则不同，它是可以像普通的中断一样被悬起的（不像 SVC 那样）。OS 可以利用它“缓期执行”一个异常——直到其它重要的任务完成后才执行动作。悬起 PendSV 的方法是：手工往 NVIC 的 PendSV 悬起寄存器中写 1。悬起后，如果优先级不够高，则将缓期等待执行。

如果一个发生的异常不能被即刻响应，就称它被“悬起”(pending)。不过，少数 fault异常是不允许被悬起的。一个异常被悬起的原因，可能是系统当前正在执行一个更高优先级异常的服务例程，或者因相关掩蔽位的设置导致该异常被除能。对于每个异常源，在被悬起的情况下，都会有一个对应的“悬起状态寄存器”保存其异常请求，直到该异常能够执行为止，这与传统的 ARM 是完全不同的。在以前，是由产生中断的设备保持住请求信号。现在NVIC 的悬起状态寄存器的出现解决了这个问题，即使后来设备已经释放了请求信号，曾经的中断请求也不会错失。

系统任务切换的工程分析

在系统中正常执行的任务（假设没有外部中断 IRQ ），用 Systick 直接做上下文切换是完全没有问题的，如图：

- 系统滴答定时器（SYSTICK）中断，（轮转调度中需要）

让我们举个简单的例子来辅助理解。假设有这么一个系统，里面有两个就绪的任务，并且通过 SysTick 异常启动上下文切换。如图 7.15 所示。

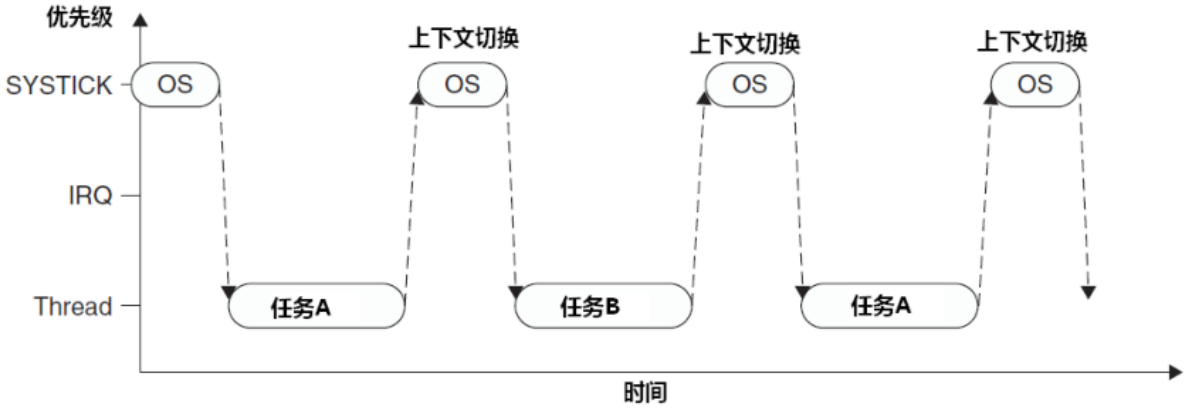

图 7.15 两个任务间通过 SysTick 轮转调度的简单模式

但是问题是几乎很少嵌入式的设备会不用其丰富的中断响应，所以，直接用systick做系统的上下文切换那是不实际的，这存在很大的风险，因为假设 `systick` 打断了一个中断（ `IRQ` ），立即做出上下文切换的话，则触犯用法 `fault` 异常，除了重启你没有其他办法了，这样子做出来的产品就是垃圾！！用我老板的话说就是写的什么狗屎！！！如图所示：

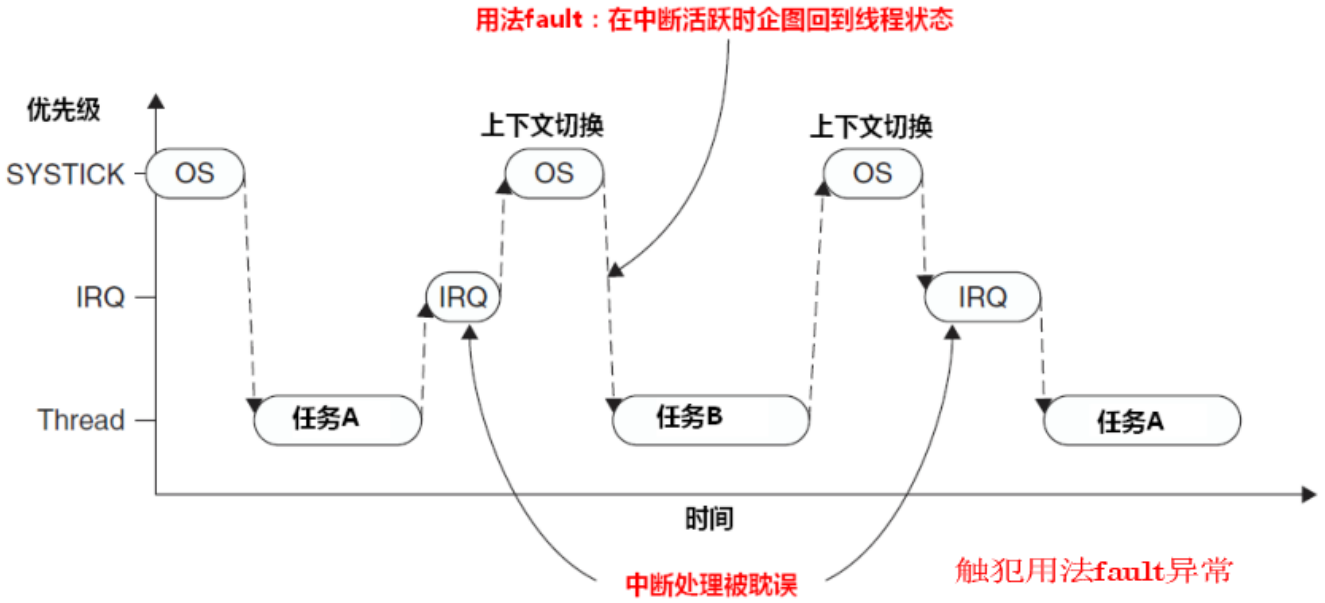

图 7.16　发生 IRQ 时上下文切换的问题 https://blog.csdn.net/jiejiemcu

那这么说这样不行那也不行，怎么办啊？请看看前面接介绍的 `PendSV` ，是不是有点豁然开朗了？ `PendSV` 来完美解决这个问题。 `PendSV` 异常会自动延迟上下文切换的请求，直到其它的 `ISR` 都完成了处理后才放行。为实现这个机制，需要把 `PendSV` 编程为最低优先级的异常。如果 `OS` 检测到某 `IRQ` 正在活动并且被 SysTick 抢占，它将悬起一个 `PendSV` 异常，以便缓期执行上下文切换。

懂了吗？就是说，只要将 `PendSV` 的优先级设为最低的，systick即使是打断了IRQ，它也不会马上进行上下文切换，而是等到IRQ执行完， `PendSV` 服务例程才开始执行，并且在里面执行上下文切换。过程如图所示：

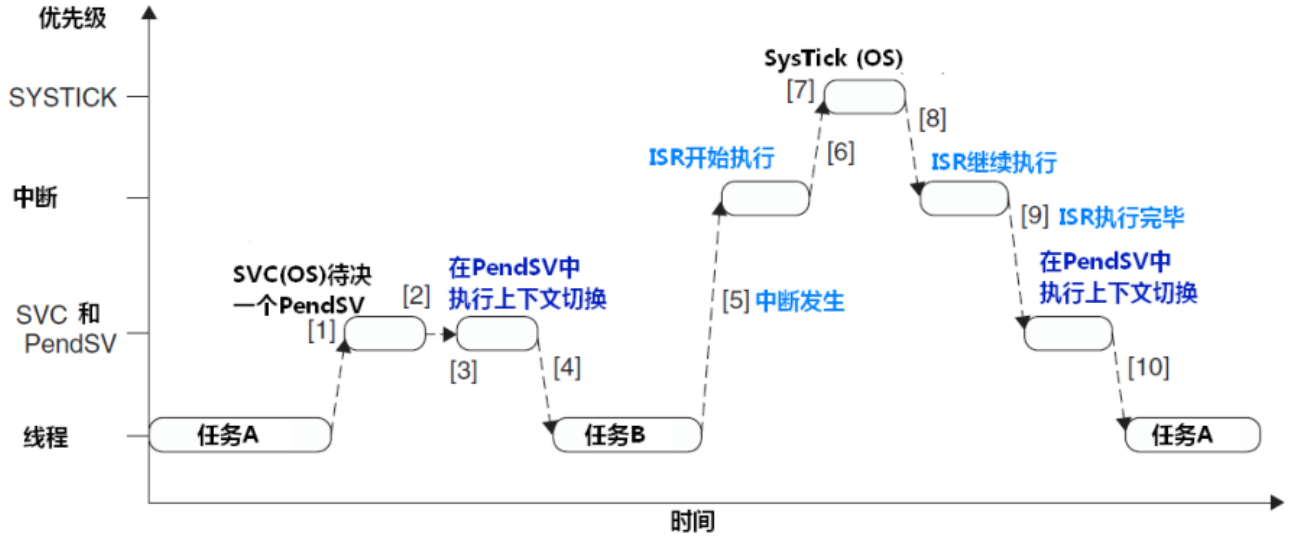

图 7.17　使用 PendSV 控制上下文切换 https://blog.csdn.net/jiejiemcu

任务切换的源码实现

过程差不多了解了，那看看FreeRTOS中怎么实现吧！！

FreeRTOS有两种方法触发任务切换：

1. 一种就是 `systick` 触发 `PendSV` 异常，这是最经常使用的。

2. 另一种是主动进行切换任务，执行系统调用，比如普通任务可以使用taskYIELD()强制任务切换，中断服务程序中使用 `portYIELD_FROM_ISR()` 强制任务切换。

第一种

先说说第一种吧，就在 `systick` 中断中调用 `xPortSysTickHandler()` ；

下面是源码：

```
void xPortSysTickHandler( void )
{
    vPortRaiseBASEPRI();
    {
        /* Increment the RTOS tick. */
        if( xTaskIncrementTick() != pdFALSE )
        {
            /* A context switch is required.  Context switching is performed in
            the PendSV interrupt.  Pend the PendSV interrupt. */
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
        }
    }
    vPortClearBASEPRIFromISR();
}
```

它的执行过程是这样子的，屏蔽所有中断，因为SysTick以最低的中断优先级运行，所以当这个中断执行时所有中断必须被屏蔽。vPortRaiseBASEPRI();就是屏蔽所有中断的。而且并不需要保存本次中断的值，因为systick的中断优先级是已知的，执行完直接恢复所有中断即可。

在 `xTaskIncrementTick()` 中会对 `tick` 的计数值进行自加，然后检查有没有处于就绪态的最优先级任务，如果有，则返回非零值，然后表示需要进行任务切换，而并非马上进行任务切换，此处要注意，它只是向中断状态寄存器 `bit28` 位写入 `1` ，只是将 `PendSV` 挂起，假如没有比 `PendSV` 更高优先级的中断，它才会进入 `PendSV` 中断服务函数进行任务切换。

```
#define portNVIC_PENDSVSET_BIT          ( 1UL << 28UL )
```

然后解除屏蔽所有中断。

```
vPortClearBASEPRIFromISR();
```

第二种

另一种方法是主动进行任务切换，不管是使用taskYIELD()还是portYIELD_FROM_ISR()，最终都会执行下面的代码：

```
#define portYIELD()                                                             \
{                                                                               \
    /* Set a PendSV to request a context switch. */                             \
    portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;                             \
                                                                                \
    __dsb( portSY_FULL_READ_WRITE );                                            \
    __isb( portSY_FULL_READ_WRITE );                                            \
}
```

这 `portYIELD()` 其实是一个宏定义来的。同样是向中断状态寄存器 `bit28位写入1` ，将 `PendSV` 挂起，然后等待任务的切换。

具体的任务切换源码

一直在说怎么进行任务切换的，好像还没看到任务切换的源码啊，哎，下面来看看任务切换的真面目！！

```
__asm void xPortPendSVHandler(void)
{
    extern uxCriticalNesting;
    extern pxCurrentTCB;
    extern vTaskSwitchContext;
```

```
    PRESERVE8
    mrs r0, psp
    isb
    ldr r3, =pxCurrentTCB       /* Get the location of the current TCB. */
    ldr r2, [r3]
    stmdb r0!, {r4-r11}         /* Save the remaining registers. */
    str r0, [r2]                /* Save the new top of stack into the first member of the TCB. */
    stmdb sp!, {r3, r14}
    mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
    msr basepri, r0
    dsb
    isb
    bl vTaskSwitchContext
    mov r0, #0
    msr basepri, r0
    ldmia sp!, {r3, r14}
    ldr r1, [r3]
    ldr r0, [r1]                /* The first item in pxCurrentTCB is the task top of stack. */
    ldmia r0!, {r4-r11}         /* Pop the registers and the critical nesting count. */
    msr psp, r0
    isb
    bx r14
    nop
}
```

不是我不想看，是我看到汇编就头大啊，这几天我也在看源码，实在是头大。

找到核心的函数看看就好啦，不管那么多，有兴趣的可以研究一下中断代码，有不懂的也很欢迎你们来问我，一起研究研究，也是不错的选择。

下面是看重点的地方了：

```
mov r0,             #configMAX_SYSCALL_INTERRUPT_PRIORITY
msr basepri, r0
```

这两句代码是关闭中断的。关中断就得干活了，嘿嘿嘿~

```
bl vTaskSwitchContext
```

BL是跳转指令嘛，这个我还是有点懂的。

调用函数 `vTaskSwitchContext()` ,寻找新的任务运行,通过使变量 `pxCurrentTCB` 指向新的任务来实现任务切换,然后就是打开中断，退出去了。

寻找下一个要运行任务

是不是感觉没什么大不了的样子，如果你是这样子觉得的，可能还没学到家，赶紧去看看 `FreeRTOS` 的源码，在 `config.h` 配置文件中是不是有一个叫做硬件查找下一个运行的任务呢？ `configUSE_PORT_OPTIMISED_TASK_SELECTION` ，这个在 `FreeRTOS` 中叫做特殊方法，其实也是硬件查找啦，但是并不是每种单片机都支持的，如果是不支持的话，只能选择软件查找的方法了，就是所谓的通用方法。通用方法我就不多说了，因为我用的是 `STM32` ，他是 支持硬件方法 的，这样子效率更高，所以我也没必要去研究他的软件方法，假如有兴趣的小伙伴可以研读一下源码，有不懂的可以向我提问，源码如下：

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()                                                  \
    {                                                                                       \
    UBaseType_t uxTopPriority = uxTopReadyPriority;                                         \
                                                                                            \
        /* Find the highest priority queue that contains ready tasks. */                    \
        while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopPriority ] ) ) )               \
        {                                                                                   \
            configASSERT( uxTopPriority );                                                  \
            --uxTopPriority;                                                                \
        }                                                                                   \
                                                                                            \
```

```
        /* listGET_OWNER_OF_NEXT_ENTRY indexes through the list, so the tasks of                    \
        the same priority get an equal share of the processor time. */                              \
        listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopPriority ] ) );       \
        uxTopReadyPriority = uxTopPriority;                                                         \
    } /* taskSELECT_HIGHEST_PRIORITY_TASK */
```

而硬件的方法源码则在下面：

```
    #define taskSELECT_HIGHEST_PRIORITY_TASK()                                                      \
    {                                                                                               \
        UBaseType_t uxTopPriority;                                                                  \
                                                                                                    \
        /* Find the highest priority list that contains ready tasks. */                            \
        portGET_HIGHEST_PRIORITY( uxTopPriority, uxTopReadyPriority );                              \
        configASSERT( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[ uxTopPriority ] ) ) > 0 );     \
        listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopPriority ] ) );       \
    } /* taskSELECT_HIGHEST_PRIORITY_TASK() */
```

其方法是利用硬件提供的计算前导零指令CLZ，具体宏定义为：

```
#define portGET_HIGHEST_PRIORITY( uxTopPriority, uxReadyPriorities ) uxTopPriority = ( 31UL - ( uint32_t ) __clz( ( uxReadyPriorities ) ) )
```

静态变量 `uxTopReadyPriority` 包含了处于就绪态任务的最高优先级的信息，因为 `FreeRTOS` 运行的永远是处于最高优先级的运行态，而下个处于最高优先级的就绪态则必定会在下次任务切换的时候运行， `uxTopReadyPriority` 使用每一位来表示任务是否处于就绪态，比如变量 `uxTopReadyPriority` 的 `bit0为1` ，则表示存在优先级为0的任务处于就绪态， `bit6为1` 则表示存在优先级为6的任务处于就绪态。并且，由于 `bit0` 的优先级高于 `bit6` ，那么下个任务就是bit0的任务运行了（数组越低优先级越高）。由于32位整形数最多只有 `32` 位，因此使用这种特殊方法限定最大可用优先级数目为 `32` ，即优先级 `0~31` 。得到了下个处于最高优先级就绪态任务了，就调用 `listGET_OWNER_OF_NEXT_ENTRY` 来获取下一个任务的列表项，然后将该列表项的任务控制块TCB赋值给 `pxCurrentTCB` ，那么我们就得到下一个要运行的任务了。

至此，任务切换已经完成。