FreeRTOS任务切换过程深层解析

FreeRTOS　系统的任务切换最终都是在 PendSV 中断服务函数中完成的，uCOS 也是在 PendSV 中断中完成任务切换的。

【为什么用PendSV异常来做任务切换】

PendSV 可以像普通中断一样被 Pending（往 NVIC　的 PendSV 的 Pend 寄存器写 1），常用的场合是 OS 进行上下文切换；它可以手动拉起后，等到比他优先级更高的中断完成后，再

假设，带 OS 系统的 CM3 中有两个就绪的任务，上下文切换可以发生在 SYSTICK 中断中：

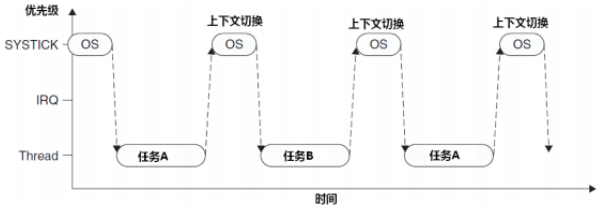


图 7.15　两个任务间通过 SysTick 进行轮转调度的简单模式

这里展现的是两个任务 A 和 B 轮转调度的过程；但是，如果在产生 SYSTICK 异常时，系统正在响应一个中断，则 SYSTICK 异常会抢占其他 ISR。在这种情况下 OS 是不能执行上下文切
请求被延迟；

而且，如果在 SYSTICK 中做任务切换，那么就会尝试切入线程模式，将导致用法 fault 异常；因为正常来说，即使SYSTICK优先级比IRQ高，当SYSTICK执行完后，也应该是回到低优先
而不是直接切换到任务去运行了，这时候IRQ都没运行完呢，怎么能中断函数都没处理完就去运行主程序呢？自然是不允许这样的。

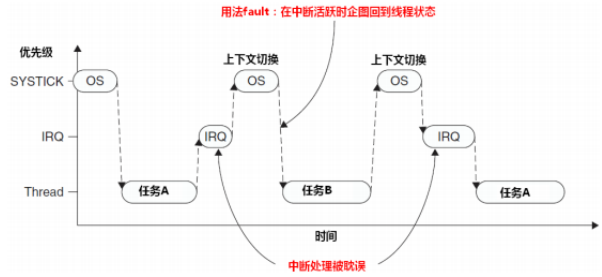


图 7.16　发生 IRQ 时上下文切换的问题

为了解决这种问题，早期的 OS 在上下文切换的时候，检查是否有中断需要响应，没有的话，采取切换上下文，然而这种方法的问题在于，可能会将任务切换的动作拖延很久（如果此次的
文，那么要等到下一次 SYSTICK 再来切换），严重的情况下，如果某 IRQ 来的频率和 SYSTICK 来的频率比较接近的时候，会导致上下文切换迟迟得不到进行；
引入 PendSV 以后，可以将 PendSV 的异常优先级设置为最低，在 PendSV 中去切换上下文，PendSV 会在其他 ISR 得到相应后，立马执行：

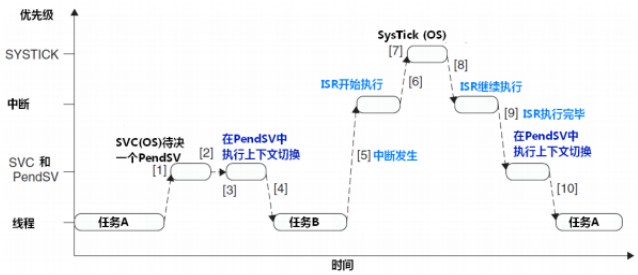


图 7.17　使用 PendSV 控制上下文切换

上图的过程可以描述为：

1、任务 A 呼叫 SVC 请求任务切换；

2、OS 收到请求，准备切换上下文，手动 Pending 一个 PendSV；

3、CPU 退出 SVC 的 ISR 后，发现没有其他 IRQ 请求，便立即进入 PendSV 执行上下文切换；

4、正确的切换到任务 B；

5、此刻发生了一个中断，开始执行此中断的 ISR；

6、ISR 执行一半，SYSTICK 来了，抢占了该 IRQ；

7、OS 执行一些逻辑，并手动 Pending PendSV 准备上下文切换；

8、退出 SYSTICK 的 ISR 后，由于之前的 IRQ 优先级高于 PendSV，所以之前的 ISR 继续执行；

9、ISR 执行完毕退出，此刻没有优先级更高的 IRQ，那么执行 PendSV 进行上下文切换；

10、PendSV 执行完毕，顺利切到任务 A，同时进入线程模式；

以上部分摘自：https://www.cnblogs.com/god-of-death/p/14856578.html

【如何设定PendSV优先级】

表D.16 系统异常优先级寄存器 0xE000_ED18 - 0xE000_ED23

地址	名称	类型	复位值	描述
0xE000_ED18	PRI_4			存储器管理 fault 的优先级
0xE000_ED19	PRI_5			总线 fault 的优先级
0xE000_ED1A	PRI_6			用法 fault 的优先级
0xE000_ED1B	-	-	-	-
0xE000_ED1C	-	-	-	-
0xE000_ED1D	-	-	-	-
0xE000_ED1E	-	-	-	-
0xE000_ED1F	PRI_11			SVC 优先级
0xE000_ED20	PRI_12			调试监视器的优先级
0xE000_ED21	-	-	-	-
0xE000_ED22	PRI_14			PendSV 的优先级
0xE000_ED23	PRI_15			SysTick 的优先级

往地址为0xE000ED22的寄存器PRI_14写入PendSV优先级

```
NVIC_SYSPRI14   EQU     0xE000ED22
NVIC_PENDSV_PRI EQU           0xFF

    LDR     R1, =NVIC_PENDSV_PRI
    LDR     R0, =NVIC_SYSPRI14
    STRB    R1, [R0]  ;将r1 中的 [7:0]存储到 r0 对应的内存
    BX      LR   ;返回
```

【如何触发PendSV异常】

表8.5 中断控制及状态寄存器ICSR（地址：0xE000_ED04）

位段	名称	类型	复位值	描述
31	NMIPENDSET	R/W	0	写 1 以悬起 NMI。因为 NMI 的优先级最高且从不掩蔽，在置位此位后将立即进入 NMI 服务例程。
28	PENDSVSET	R/W	0	写 1 以悬起 PendSV。读取它则返回 PendSV 的状态
27	PENDSVCLR	W	0	写 1 以清除 PendSV 悬起状态
26	PENDSTSET	R/W	0	写 1 以悬起 SysTick。读取它则返回 PendSV 的状态

往ICSR第28位写1，即可将PendSV异常挂起。若是当前没有高优先级中断产生，那么程序将会进入PendSV handler

```
NVIC_INT_CTRL   EQU     0xE000ED04
NVIC_PENDSVSET  EQU     0x10000000

    LDR     R0, =NVIC_INT_CTRL
    LDR     R1, =NVIC_PENDSVSET
    STR     R1, [R0]
    BX      LR
```

【测试PendSV异常handler实现任务切换】

如何实现任务切换？三个步骤：

步骤一：在进入中断前先设置PSP。

步骤二：将当前寄存器的内容保存到当前任务堆栈中。进入ISR时，cortex-m3会自动保存八个寄存器到PSP中，剩下的几个需要我们手动保存。

步骤三：在Handler中将下一个任务的堆栈中的内容加载到寄存器中，并将PSP指向下一个任务的堆栈。这样就完成了任务切换。

要在PendSV 的ISR中完成这两个步骤，我们先需了解下在进入PendSV ISR时，cortex-M3做了什么？

1，入栈。会有8个寄存器自动入栈。入栈内容及顺序如下：

表9.1 入栈顺序以及入栈后堆栈中的内容

地址	寄存器	被保存的顺序
旧SP (N-0)	原先已压入的内容	-
(N-4)	xPSR	2
(N-8)	PC	1
(N-12)	LR	8
(N-16)	R12	7
(N-20)	R3	6
(N-24)	R2	5
(N-28)	R1	4
新SP (N-32)	R0	3

在步骤一中，我们已经设置了PSP，那这8个寄存器就会自动入栈到PSP所指地址处。

2，取向量。找到PendSV ISR的入口地址，这样就能跳到ISR了。

3，更新寄存器内容。

做完这三步后，程序就进入ISR了。

进入ISR前，我们已经完成了步骤一，cortex-M3已经帮我们完成了步骤二的一部分，剩下的需要我们手动完成。

在ISR中添加代码如下：

MRS R0, PSP

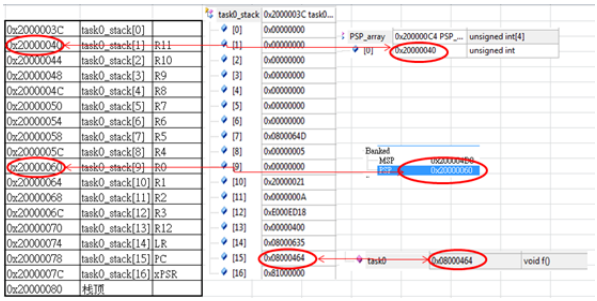
保存PSP到R0。为什么是PSP而不是MSP。因为在OS启动的时候，我们已经把SP设置为PSP了。这样使得用户程序使用任务堆栈，OS使用主堆栈，不会互相干扰。不会因为用户程序导

STMDB R0!,{R4-R11}

保存R4-R11到PSP中。C语言表达是*(–R0)={R4-R11}，R0中值先自减1，然后将R4-R11的值保存到该值所指向的地址中，即PSP中。

STMDB Rd!，{寄存器列表} 连续存储多个字到Rd中的地址值所指地址处。每次存储前，Rd先自减一次。

若是ISR是从从task0进来，那么此时task0的堆栈中已经保存了该任务的寄存器参数。保存完成后，当前任务堆栈中的内容如下(假设是task0)



左边表格是预期值，右边是keil调试的实际值。可以看出，是一致的。在任务初始化时(步骤一)，我们将PSP指向任务0的栈顶0x20000080。在进入PendSV之前，cortex-M3自动入栈八个

0x20000060。然后我们再保存R4-R11到0x20000040~0x2000005C。

这样很容易看明白，如果需要下次再切换到task0，只需恢复R4~R11，再将PSP指向0x20000060即可。

测试例程：

```
#define HW32_REG(ADDRESS)  (*((volatile unsigned long  *)(ADDRESS)))
void USART1_Init(void);
void task0(void) ;

uint32_t  curr_task=0;     // 当前执行任务
uint32_t  next_task=1;     // 下一个任务
uint32_t task0_stack[17];
uint32_t task1_stack[17];
uint32_t  PSP_array[4];

u8 task0_handle=1;
u8 task1_handle=1;

void task0(void)
{
    while(1)
    {
        if(task0_handle==1)
        {
            printf("task0\n");
            task0_handle=0;
            task1_handle=1;
        }
    }
}

void task1(void)
{
    while(1)
    {
        if(task1_handle==1)
        {
            printf("task1\n");
            task1_handle=0;
            task0_handle=1;
        }
    }
}

__asm void SetPendSVPro(void)
{
NVIC_SYSPRI14   EQU     0xE000ED22
NVIC_PENDSV_PRI EQU           0xFF

    LDR     R1, =NVIC_PENDSV_PRI
    LDR     R0, =NVIC_SYSPRI14
    STRB    R1, [R0]
    BX      LR
}

__asm void TriggerPendSV(void)
{
NVIC_INT_CTRL   EQU     0xE000ED04
NVIC_PENDSVSET  EQU     0x10000000

    LDR     R0, =NVIC_INT_CTRL
    LDR     R1, =NVIC_PENDSVSET
    STR     R1, [R0]
    BX      LR
}

int main(void)
```

```
{
    SetPendSVPro();
    LED_Init();
    uart_init(115200);

    printf("OS test\n");

    PSP_array[0] = ((unsigned int) task0_stack) + (sizeof task0_stack) - 16*4;
    //PSP_array中存储的为task0_stack数组的尾地址-16*4, 即task0_stack[1]的地址
    HW32_REG((PSP_array[0] + (14*4))) = (unsigned long) task0; /* PC */
    //task0的PC存储在task0_stack[1]地址+14*4, 即task0_stack[15]的地址中
    HW32_REG((PSP_array[0] + (15*4))) = 0x01000000;            /* xPSR */

    PSP_array[1] = ((unsigned int) task1_stack) + (sizeof task1_stack) - 16*4;
    HW32_REG((PSP_array[1] + (14*4))) = (unsigned long) task1; /* PC */
    HW32_REG((PSP_array[1] + (15*4))) = 0x01000000;            /* xPSR */

    /* 任务0先执行 */
    curr_task = 0;

    /* 设置PSP指向任务0堆栈的栈顶 */
    __set_PSP((PSP_array[curr_task] + 16*4));

    SysTick_Config(9000000);
    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);//72/8=9MHZ
    /* 使用堆栈指针, 非特权级状态 */
    __set_CONTROL(0x3);

    /* 改变CONTROL后执行ISB (architectural recommendation) */
    __ISB();

    /* 启动任务0 */
    task0();
    //LED0=0;
    while(1);
}

__asm void PendSV_Handler(void)
{
    // 保存当前任务的寄存器内容
    MRS    R0, PSP     // 得到PSP  R0 = PSP
                       // xPSR, PC, LR, R12, R0-R3已自动保存
    STMDB  R0!,{R4-R11}// 保存R4-R11共8个寄存器得到当前任务堆栈

    // 加载下一个任务的内容
    LDR    R1,=__cpp(&curr_task)
    LDR    R3,=__cpp(&PSP_array)
    LDR    R4,=__cpp(&next_task)
    LDR    R4,[R4]     // 得到下一个任务的ID
    STR    R4,[R1]     // 设置 curr_task = next_task
    LDR    R0,[R3, R4, LSL #2] // 从PSP_array中获取PSP的值
    LDMIA  R0!,{R4-R11}// 将任务堆栈中的数值加载到R4-R11中
  //ADDS   R0, R0, #0x20
    MSR    PSP, R0     // 设置PSP指向此任务
 // ORR     LR, LR, #0x04
    BX     LR          // 返回
                       // xPSR, PC, LR, R12, R0-R3会自动的恢复
    ALIGN  4
}

void SysTick_Handler(void)
{
    LED0=!LED0;//位带操作
    if(curr_task==0)
        next_task=1;
    else
        next_task=0;
    TriggerPendSV();
}
```

串口输出：

```
OS test
task0
task1
task0
task1
task0
task1
task0
task1
task0
task1
task0
task1
task0
task1
task0
```

可以看到在任务0和任务1之间来回切换。

【FreeRTOS任务切换源码】

上下文(任务)切换被触发的场合大致分为：
● 可以执行一个系统调用
● 系统滴答定时器(SysTick)中断。

执行系统调用就是执行 FreeRTOS 系统提供的相关 API 函数，比如任务切换函数taskYIELD()，这些 API 函数和任务切换函数taskYIELD()都统称为系统调用。

函数 taskYIELD()其实就是个宏，在文件 task.h 中有如下定义：
#define taskYIELD() portYIELD()

函数 portYIELD()也是个宏，在文件 portmacro.h 中有如下定义

```
#define portYIELD()  \
{  \
portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT; \   //通过向中断控制和壮态寄存器 ICSR 的 bit28 写入 1 挂起 PendSV 来启动 PendSV 中断。这样就可以在 PendSV 中断服务函数中进
\
__dsb( portSY_FULL_READ_WRITE );  \
__isb( portSY_FULL_READ_WRITE );  \
}
```

中断级的任务切换函数为 portYIELD_FROM_ISR()，定义如下：

```
#define portYIELD_FROM_ISR( x ) portEND_SWITCHING_ISR( x )

#define portEND_SWITCHING_ISR( xSwitchRequired ) \
    if( xSwitchRequired != pdFALSE ) portYIELD()   //可以看出 portYIELD_FROM_ISR()最终也是通过调用函数 portYIELD()来完成任务切换的。
```

系统滴答定时器(SysTick)中断

```
void SysTick_Handler(void)
{
    if(xTaskGetSchedulerState()!=taskSCHEDULER_NOT_STARTED)//系统已经运行
    {
        xPortSysTickHandler();
    }
}
```

xPortSysTickHandler()源码如下：

```
void xPortSysTickHandler( void )
{
    vPortRaiseBASEPRI(); //关闭中断
    {
        if( xTaskIncrementTick() != pdFALSE )  //增加时钟计数器 xTickCount 的值
        {
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT; //通过向中断控制和壮态寄存器 ICSR 的 bit28 写入 1 挂起 PendSV 来启动 PendSV 中断。这样就可以在 PendSV 中断服
        }
    }
    vPortClearBASEPRIFromISR(); //打开中断
}
```

真正的任务切换代码在PendSV中断函数中，
FreeRTOS做了如下函数重定义
#define xPortPendSVHandler PendSV_Handler

xPortPendSVHandler函数如下 (汇编 port.c)

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

    PRESERVE8

    mrs r0, psp  //读取进程栈指针，保存在寄存器 R0 里面。
    isb

    ldr r3, =pxCurrentTCB //获取当前任务的任务控制块
    ldr r2, [r3]  //接上，并将任务控制块的地址保存在寄存器 R2 里面

    tst r14, #0x10 //判断任务是否使用了 FPU，如果任务使用了 FPU 的话在进行任务切换的时候就
    it eq          //需要将 FPU 寄存器 s16~s31 手动保存到任务堆栈中，其中 s0~s15 和 FPSCR 是自动保存的

    vstmdbeq r0!, {s16-s31} //保存 s16~s31 这 16 个 FPU 寄存器
    stmdb r0!, {r4-r11, r14} //保存 r4~r11 和 R14 这几个寄存器的值
    str r0, [r2]  //将寄存器 R0 的值写入到寄存器 R2 所保存的地址中去，也就是将新的栈顶保存在任务控制块的第一个字段中。
    stmdb sp!, {r3} //将寄存器 R3 的值临时压栈，寄存器 R3 中保存了当前任务的任务控制块
    mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY //关闭中断，进入临界区
    msr basepri, r0                               //关闭中断，进入临界区
    dsb
    isb
    bl vTaskSwitchContext //调用函数 vTaskSwitchContext()，此函数用来获取下一个要运行的任务，并将pxCurrentTCB 更新为这个要运行的任务
    mov r0, #0  //打开中断，退出临界区。
    msr basepri, r0  //打开中断，退出临界区。
    ldmia sp!, {r3} //刚刚保存的寄存器 R3 的值出栈，恢复寄存器 R3 的值
    ldr r1, [r3]  //获取新的要运行的任务的任务堆栈栈顶，
    ldr r0, [r1]  //接上，并将栈顶保存在寄存器 R0 中
```

```
    ldmia r0!, {r4-r11, r14} //R4~R11,R14 出栈，也就是即将运行的任务的现场
    tst r14, #0x10 //判断即将运行的任务是否有使用到 FPU，如果有的话还需要手工恢复 FPU的 s16~s31 寄存器。
    it eq          //同上
    vldmiaeq r0!, {s16-s31} //同上
    msr psp, r0 //更新进程栈指针 PSP 的值
    isb
    bx r14  //执行此行代码以后硬件自动恢复寄存器 R0~R3、R12、LR、PC 和 xPSR 的值，确定
            //异常返回以后应该进入处理器模式还是进程模式，使用主栈指针(MSP)还是进程栈指针(PSP)。
            //很明显这里会进入进程模式，并且使用进程栈指针(PSP)，寄存器 PC 值会被恢复为即将运行的
            //任务的任务函数，新的任务开始运行！至此，任务切换成功。
}
```