FreeRTOS Scheduler

FreeRTOS的调度原理和内核相关，因此也需要有一些Cortex-M内核相关的知识。

一些概念

- 在Cortex-M内核中，使用Systick作为心跳时钟，一般默认是1ms。
- 进入Systick中断后，内核会在系统的就绪列表中从高优先级开始找需要执行的任务，如果任务状态发生了变化，就会产生一个PendSV中断，内核会在PendSV中断中,改变进程的栈指针PSP，进行任务切换。

Systick

对于一个系统来说，时钟是系统能够运转的核心，也就是调度器的核心了。因此，我们可以来研究一下Systick

在FreeRTOS中，Systick的初始化函数为：vPortSetupTimerInterrupt，也就是：

```
void vPortSetupTimerInterrupt( void )
{
    /* Calculate the constants required to configure the tick interrupt. */
    #if( configUSE_TICKLESS_IDLE == 1 )
    {
        ulTimerCountsForOneTick = ( configSYSTICK_CLOCK_HZ / configTICK_RATE_HZ );
        xMaximumPossibleSuppressedTicks = portMAX_24_BIT_NUMBER /
ulTimerCountsForOneTick;
        ulStoppedTimerCompensation = portMISSED_COUNTS_FACTOR / (
configCPU_CLOCK_HZ / configSYSTICK_CLOCK_HZ );
    }
    #endif

    portNVIC_SYSTICK_CTRL_REG = 0UL;
    portNVIC_SYSTICK_CURRENT_VALUE_REG = 0UL;

    portNVIC_SYSTICK_LOAD_REG = ( configSYSTICK_CLOCK_HZ / configTICK_RATE_HZ ) -
1UL;
    portNVIC_SYSTICK_CTRL_REG = ( portNVIC_SYSTICK_CLK_BIT |
portNVIC_SYSTICK_INT_BIT | portNVIC_SYSTICK_ENABLE_BIT );
}
```

而Systick的中断服务函数是：xPortSysTickHandler：

```
void xPortSysTickHandler( void )
{
    portDISABLE_INTERRUPTS();
    {
        if( xTaskIncrementTick() != pdFALSE )
        {
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
        }
```

```
    }
    portENABLE_INTERRUPTS();
}
```

可以看到 中断服务函数里面使用了xTaskIncrementTick()函数，好好研究一下，该函数原型：

```
BaseType_t xTaskIncrementTick( void )
{
    /* 1. 创建一个临时任务块 */
    TCB_t * pxTCB;

    /* 2. 定义了一个存储某个任务状态列表项里的值，这个值通常表示任务的延迟时间或者超时时间，也就是任务要等待多少个滴答数才从阻塞转为就绪 */
    TickType_t xItemValue;

    /* 3. 定义了一个布尔类型的变量，用于表示是否需要进行任务切换 */
    BaseType_t xSwitchRequired = pdFALSE;

    /* 4. 记录更新当前的时钟节拍数 */
    traceTASK_INCREMENT_TICK( xTickCount );

    /* 5. 用来判断调度器是否被挂起，因为当调度器被挂起的时候，不会发生任务切换 */
    if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )
    {

    /* 6. 如果没有被挂起 则把计时器加一，表示下一个节拍 */
        const TickType_t xConstTickCount = xTickCount + ( TickType_t ) 1;

    /* 7. 更新当前的时钟节拍数 */
        xTickCount = xConstTickCount;

        /* 8. 如果节拍数等于0 */
        if( xConstTickCount == ( TickType_t ) 0U )
        {

            /* 9. 切换延迟任务列表 */
            taskSWITCH_DELAYED_LISTS();
        }
        else
        {
            /* 10. 否则标记测试覆盖率 */
            mtCOVERAGE_TEST_MARKER();
        }

        /* 11. 如果当前的节拍数大于或等于下一个任务解除阻塞的时间 */
        if( xConstTickCount >= xNextTaskUnblockTime )
        {
            for( ;; )
            {
                /* 12. 检查延迟任务列表是否为空 */
                if( listLIST_IS_EMPTY( pxDelayedTaskList ) != pdFALSE )
```

```
                {
                    /* 13. 如果延迟任务列表为空，则下一个任务的阻塞时间设置为无限大 */
                    xNextTaskUnblockTime = portMAX_DELAY;
                    break;
                }
                else
                {
                    /* 13. 从延迟任务列表中获取头部任务的控制块，赋值给临时任务控制块，
主要是来将延迟状态转换为就绪态 */
                    pxTCB = listGET_OWNER_OF_HEAD_ENTRY( pxDelayedTaskList );

                    /* 14. 获取任务的延迟时间,存到临时变量里面 */
                    xItemValue = listGET_LIST_ITEM_VALUE( &( pxTCB->xStateListItem
) );

                    /* 15. 如果当前的节拍数小于任务的延迟时间，说明任务还不能被解除阻塞
*/
                    if( xConstTickCount < xItemValue )
                    {

                        /* 16. 说明任务还没有到时间，需要跳出此次循环，继续遍历其他任务
*/
                        xNextTaskUnblockTime = xItemValue;
                        break;
                    }d
                    else
                    {

                        mtCOVERAGE_TEST_MARKER();
                    }

                    /* 17. 从当前列表项里把该任务去除，也就是把任务从阻塞转到就绪 */
                    ( void ) uxListRemove( &( pxTCB->xStateListItem ) );

                    /* 18. 如果任务的事件列表项已经有了拥有者也就是处于某个列表中 */
                    if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) !=
NULL )
                    {

                        /* 19. 从任务的事件列表项中把该任务去除，也就是把任务从阻塞转到
就绪 */
                        ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }

                    /* 21. 把任务添加到就绪列表中 */
                    prvAddTaskToReadyList( pxTCB );

                    /* 22. 如果定义了是抢占式调度 */
                    #if (  configUSE_PREEMPTION == 1 )
                    {
```

```
                        /* 23. 并且任务的优先级大于当前任务的优先级，需要进行任务切换
*/
                        if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )
                        {
                            /* 24. 标记需要进行任务切换 */
                            xSwitchRequired = pdTRUE;
                        }
                        else
                        {
                            mtCOVERAGE_TEST_MARKER();
                        }
                    }
                    #endif /* configUSE_PREEMPTION */
                }
            }
        }

        /* 25. 如果宏定义了抢占式调度和启用时间片轮询调度 */
        #if ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING == 1 ) )
        {
            /* 26. 如果当前优先级的任务的就绪列表里任务数量大于1，那么设置
xSwitchRequired为true,这表明有多个任务可以切换 */
            if( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[ pxCurrentTCB-
>uxPriority ] ) ) > ( UBaseType_t ) 1 )
            {
                xSwitchRequired = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING == 1
) ) */

        /* 27. 如果宏定义了使用空闲钩子 */
        #if ( configUSE_TICK_HOOK == 1 )
        {

            /* 28. 并且当前的节拍数等于0，调用钩子函数 */
            if( xPendedTicks == ( TickType_t ) 0 )
            {
                vApplicationTickHook();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* configUSE_TICK_HOOK */

        /* 29. 如果配置了抢占式调度 */
        #if ( configUSE_PREEMPTION == 1 )
```

```
        {
            /* 30. 有一个任务已经请求放弃 CPU */
            if( xYieldPending != pdFALSE )
            {
                /* 31. 切换任务 */
                xSwitchRequired = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* configUSE_PREEMPTION */
    }
    /* 32. 如果调度器被挂起，那么就记录下有多少个节拍数被挂起 */
    else
    {
        /* 33. 滴答数++ */
        ++xPendedTicks;

        /* 34. 如果配置了使用钩子函数，就进入空闲钩子函数 */
        #if ( configUSE_TICK_HOOK == 1 )
        {
            vApplicationTickHook();
        }
        #endif
    }

    /* 35. 返回是否需要进行任务切换 */
    return xSwitchRequired;
}
```

因此，我们知道了在Systick的中断服务函数xPortSysTickHandle里，可以决定着要不要令 portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT

如果xSwitchRequired为true，则把portNVIC_INT_CTRL_REG写入portNVIC_PENDSVSET_BIT，来触发PendSV 异常。，而PendSV异常与任务的切换有着紧密关系。

- 当PendSV异常发生时，会调用中断服务函数也就是xPortPendSVHandler。

那么现在就来看一下PendSV异常的处理函数：

PendSV

xPortPendSVHandler函数原型：这是一段汇编

```
void xPortPendSVHandler( void )
{
    /* This is a naked function. */

    __asm volatile
```

```
    (
    /* 1. 将当前的任务堆栈指针的值移动到R0寄存器中，PSP是当前任务的栈顶指针 */
    "   mrs r0, psp                         \n"

    /* 2. 指令同步屏障，确保之前的指令执行完成 */
    "   isb                                 \n"

    /* 3. 加载当前任务控制块的地址到寄存器R3中 */
    "   ldr r3, pxCurrentTCBConst           \n" /* Get the location of the current
TCB. */

    /* 4. 加载当前任务控制块的地址到寄存器R2中 */
    "   ldr r2, [r3]                        \n"

    /* 5. 任务是否使用了FPU TST指令就是比较测试两个寄存器的值*/
    "   tst r14, #0x10                      \n" /* Is the task using the FPU
context?  If so, push high vfp registers. */

    /* 6. 如果使用了FPU，那么就把S16-S31寄存器的值压入堆栈中 (it指令就是if-then,eq就是
equal，vstmdbeq是浮点寄存器)*/
    "   it eq                               \n"
    "   vstmdbeq r0!, {s16-s31}             \n"

    /* 7. 将核心寄存器 (r4-r11) 和链接寄存器 (r14) 保存到 r0 指向的地址 */
    "   stmdb r0!, {r4-r11, r14}            \n" /* Save the core registers. */

    /* 8. 将新的堆栈顶部地址保存到 TCB 的第一个成员中，这个成员指向当前任务的堆栈顶部*/
    "   str r0, [r2]                        \n" /* Save the new top of stack into
the first member of the TCB. */

    /* 9. 将 r0 和 r3 寄存器的值压入系统堆栈指针(Main Stack Pointer，MSP) */
    "   stmdb sp!, {r0, r3}                 \n"

    /* 10. 将 basepri 的值(即 configMAX_SYSCALL_INTERRUPT_PRIORITY)移动到 r0 中 */
    "   mov r0, %0                          \n"

    /* 11. 设置 basepri (基优先级) 寄存器的值，以确保 PendSV 处理期间不允许更高优先级的
中断打断 */
    "   msr basepri, r0                     \n"\

    /* 12. 数据同步屏障和指令同步屏障，确保之前的写操作完成 */
    "   dsb                                 \n"
    "   isb                                 \n"

    /* 13. 调用 vTaskSwitchContext 函数，这个函数会切换到下一个任务,更新TCB */
    "   bl vTaskSwitchContext               \n"

    /* 14. 将 basepri 寄存器重置为 0，以允许所有优先级的中断 */
    "   mov r0, #0                          \n"
    "   msr basepri, r0                     \n"

    /* 15. 从系统堆栈中弹出 r0 和 r3 的值 */
    "   ldmia sp!, {r0, r3}                 \n"
```

```
    /* 16. 从 r3 指向的地址加载 TCB 的第一个成员到 r1 中 */
    "   ldr r1, [r3]                        \n" /* The first item in pxCurrentTCB
is the task top of stack. */

    /* 17. 从 r1 指向的地址加载 TCB 的值到 r0 中 */
    "   ldr r0, [r1]                        \n"

    /* 18. 从下一个任务的堆栈顶部弹出核心寄存器和链接寄存器的值 */
    "   ldmia r0!, {r4-r11, r14}            \n" /* Pop the core registers. */

    /* 19. 如果使用了FPU，那么就把S16-S31寄存器的值弹出 */
    "   tst r14, #0x10                      \n" /* Is the task using the FPU
context?  If so, pop the high vfp registers too. */
    "   it eq                               \n"
    "   vldmiaeq r0!, {s16-s31}             \n"

    /* 20. 将下一个任务的堆栈顶部地址设置为 PSP，以便任务恢复执行 */
    "   msr psp, r0                         \n"

    /* 21. 指令同步屏障，确保 PSP 的更新完成。 */
    "   isb                                 \n"


    #ifdef WORKAROUND_PMU_CM001
        #if WORKAROUND_PMU_CM001 == 1
    "           push { r14 }                \n"
    "           pop { pc }                  \n"
        #endif
    #endif

    /* 22. 跳转到R14 */
    "   bx r14                              \n"
    "                                       \n"
    "   .align 4                            \n"
    "pxCurrentTCBConst: .word pxCurrentTCB  \n"
    ::"i"(configMAX_SYSCALL_INTERRUPT_PRIORITY)
    );
}
```

现在来一步步的详细分析切换过程，上下文切换其实主要就是实现保存现场-切换任务-恢复现场：

1. mrs r0, psp

- mrs指令和LDR是一样的，只是权限更高，可以访问更多的寄存器。
- mrs r0, psp 是将当前任务的栈顶指针（PSP）的值加载到寄存器 r0 中,PSP这里的值就是pxCurrentTCB->pxTopOfStack
- 这里要注意一点，在ARM-Conrtex内核中，使用着双堆栈指针机制，也就是一个是PSP，一个是MSP，MSP是用来管理异常的，而PSP是用来管理任务的。这里虽然说PendSV中断里运行着上下文的切换，但由于PendSV的目的就是处理任务的，所以这里使用的还是PSP指针，而不是MSP指针。

2. isb

指令同步隔离，确保之前的指令执行完成。

3. ldr r3, pxCurrentTCBConst

- ldr指令，就是把后面的值读出来放到前面那个寄存器里，这里就是说把pxCurrentTCBConst的地址放到r3寄存器里

4. ldr r2, [r3]

- 指令里的[]括起来表示间接寻址，也就是把R3寄存器的值指向的内存地址中的数据读出来放到R2寄存器里
- 从 r3 寄存器指向的内存地址（即当前任务的TCB地址）中读取数据，并将其加载到寄存器 r2 中。这里的数据是任务控制块的首地址，因为 pxCurrentTCB 存储的是当前任务的TCB地址

5. tst r14, #0x10

- tst指令是一个测试两个寄存器的值，会把后面两个寄存器的值按位进行与运算，这里就是把R14和#0x10进行比较
- 在ARM架构里，LR寄存器的第4位是FPU使能位，因为当一个任务使用了FPU后，除了要保存常规的寄存器，还要保护和恢复S16-S31寄存器，也就是浮点寄存器。

6. it eq ; vstmdbeq r0!, {s16-s31}

- it eq表示的就是if-then equal 也就是如果相等，那么就执行后面的指令
- 执行的就是将s16到s31这16个浮点寄存器的值存储到由r0寄存器指向的内存地址中，并且存储后r0的值会递减，为下一个存储操作做好准备

7. stmdb r0!, {r4-r11, r14}

- R4-R11是常规的核心寄存器，R14就是LR寄存器
- 这一指令是说把R4-R11以及R14的LR寄存器的值存储到由r0寄存器指向的内存地址中，而在一开始R0寄存器中指向的是PSP也就是当前任务的栈顶指针，所以相当于把核心寄存器还有LR寄存器存到了当前任务的栈中，也就是自己的栈空间，这也是为了方便后面恢复现场。

8. str r0, [r2]

- str指令可以粗略理解为写指令。也就是把R0寄存器的值写入到R2寄存器指向的内存地址中，也就是把当前任务的栈顶指针写入到当前任务的TCB中，也就是pxCurrentTCB->pxTopOfStack = r0

9. stmdb sp!, {r0, r3}

- ARM架构里，SP指针用来指向当前任务的堆栈，这里就是把R0和R3寄存器的值写入到SP指针指向的内存地址中
- 也就是把R0(当前任务的栈顶)和R3(当前任务控制块的地址)的寄存器内容存储到由sp指向的内存地址中

10. mov r0, #0

- 把R0清空

11. msr basepri, r0

- basepri是基优先级寄存器，用于设置中断屏蔽的阈值，主要是用来确定是否允许中断被处理。
- 这个的意思就是把r0寄存器的值写入到basepri寄存器中，也就是把0写入到basepri寄存器中，也就是把basepri寄存器的值设置为0，也就是允许所有优先级的中断被处理

12. dsb

- 数据同步屏障，确保之前的写操作完成

13. isb

- 指令同步屏障，确保之前的写操作完成

14. bl vTaskSwitchContext

- bl指令是跳转到vTaskSwitchContext函数，这个函数是用来切换任务的,主要就是找出当前就绪态链表中最高优先级的任务，并将当前任务控制块 pxCurrentTCB 的值更新为这个任务的TCB，后面会详细分析

15. mov r0, #0

- 把R0清空

16. msr basepri, r0

- 把R0写入到basepri寄存器中，也就是把basepri寄存器的值设置为0，也就是允许所有优先级的中断被处理

17. ldmia sp!, {r0, r3}

- 从由sp指向的内存地址中加载R0和R3寄存器的值，也就是把SP指针指向的内存地址中的内容加载到R0和R3寄存器中，也就是把当前任务的栈顶指针和当前任务控制块的地址加载到R0和R3寄存器中

18. ldr r1, [r3]

- 从由r3指向的内存地址中加载数据，并将其加载到寄存器r1中，也就是把当前任务控制块的地址加载到R1寄存器中

19. ldr r0, [r1]

- 从由r1指向的内存地址中加载数据，并将其加载到寄存器r0中，也就是把当前任务控制块的内容加载到R0寄存器中

20. ldmia r0!, {r4-r11, r14}

- 从由r0指向的内存地址中加载数据，并将其加载到寄存器r4-r11和r14中，也就是把当前任务的栈顶指针指向的内存地址中的内容加载到R4-R11和R14寄存器中，也就是把当前任务的栈空间中的内容加载到R4-R11和R14寄存器中

21. tst r14, #0x10

- 测试R14寄存器和#0x10的值，也就是测试LR寄存器的第4位是否为1，如果为1，那么就执行后面的指令

22. it eq ; vldmiaeq r0!, {s16-s31}

- 如果R14寄存器的第4位为1，那么就执行后面的指令，也就是把R0寄存器指向的内存地址中的内容加载到s16-s31寄存器中，也就是把当前任务的栈空间中的内容加载到s16-s31寄存器中

23. msr psp, r0

- 把R0寄存器的值写入到psp寄存器中，也就是把当前任务的栈顶指针写入到psp寄存器中，也就是把psp寄存器的值设置为当前任务的栈顶指针

24. isb

- 指令同步屏障，确保之前的写操作完成

25. bx r14

- 跳转到R14寄存器指向的地址，也就是跳转到当前任务的入口函数，也就是恢复现场，开始执行当前任务

vTaskSwitchContext

上面提到PendSV中断中调用vTaskSwitchContext函数来切换任务控制块，函数原型如下：

```
void vTaskSwitchContext( void )
{
    /* 1. 调度器是否被挂起 */
    if( uxSchedulerSuspended != ( UBaseType_t ) pdFALSE )
    {
        /* 2. 被挂起，不进行任务切换 */
        xYieldPending = pdTRUE;
    }
    /* 3. 调度器没有被挂起 */
    else
    {

        xYieldPending = pdFALSE;
        traceTASK_SWITCHED_OUT();

        /* 4. 检查当前任务的堆栈是否溢出 */
        taskCHECK_FOR_STACK_OVERFLOW();

        /* 5. 选择就绪列表里面优先级最高的还没运行的任务 */
        taskSELECT_HIGHEST_PRIORITY_TASK();

        /* 6. 任务被选中后运行该宏，用于跟踪和记录当前任务 */
        traceTASK_SWITCHED_IN();
```

```
    }
}
```

可以看到，vTaskSwitchContext函数主要就是选择就绪列表里面优先级最高的还没运行的任务，然后把当前任务控制块pxCurrentTCB的值更新为这个任务的TCB。主要是在 taskSELECT_HIGHEST_PRIORITY_TASK() 这个宏里，把他展开就是：

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()
\
{
\
UBaseType_t uxTopPriority = uxTopReadyPriority;
\

\
    while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopPriority ] ) ) )
\
    {
\
        configASSERT( uxTopPriority );
\
        --uxTopPriority;
\
    }
\

    listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopPriority
] ) );          \
    uxTopReadyPriority = uxTopPriority;
\
}
```

其中会运行到listGET_OWNER_OF_NEXT_ENTRY这个宏，而再把这个宏展开：

```
#define listGET_OWNER_OF_NEXT_ENTRY( pxTCB, pxList )
\
{
\
    List_t * const pxConstList = ( pxList );
\

    ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
\
    if( ( void * ) ( pxConstList )->pxIndex == ( void * ) &( ( pxConstList )-
>xListEnd ) )   \
    {
\
        ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
\
```

```
    }
\
    ( pxTCB ) = ( pxConstList )->pxIndex->pvOwner;
\
}
```

就已经很清楚了，这里就是把当前任务控制块pxCurrentTCB的值更新为这个任务的TCB。

所以，总结一下就是Systick触发PendSV异常，PendSV异常处理函数xPortPendSVHandler会保存现场、调用vTaskSwitchContext函数，vTaskSwitchContext函数会选择就绪列表里面优先级最高的还没运行的任务，然后把当前任务控制块pxCurrentTCB的值更新为这个任务的TCB，然后再恢复现场。