FreeRTOS 任务调度 任务切换

简述

前面文章 < FreeRTOS 任务调度 任务创建 > 介绍了 FreeRTOS 中如何创建任务以及其具体实现。
一般来说， 我们会在程序开始先创建若干个任务， 而此时任务调度器还没又开始运行，因此每一次任务创建后都会依据其优先级插入到就绪链表，同时保证全局变量 `pxC` 向当前创建的所有任务中优先级最高的一个，但是任务还没开始运行。
当初始化完毕后，调用函数 `vTaskStartScheduler` 启动任务调度器开始开始调度，此时， `pxCurrentTCB` 所指的任务才开始运行。
所以， 本章，介绍任务调度器启动以及如何进行任务切换。

> 调度器涉及平台底层硬件操作，本文以Cotex-M3 架构为例， 具体可以参考 《Cortex-M3权威指南》（文末附）

分析的源码版本是 v9.0.0
(为了方便查看，github 上保留了一份源码Source目录下的拷贝)

启动调度器

创建任务后，系统不会自动启动任务调度器，需要用户调用函数 vTaskStartScheduler 启动调度器。 该函数被调用后，会先创建系统自己需要用到的任务，比如空闲任务 `t` 定时器管理的任务等。 之后， 调用移植层提供的函数 `xPortStartScheduler` 。
代码解析如下，

```
1  void vTaskStartScheduler( void )
2  {
3      BaseType_t xReturn;
4      #if( configSUPPORT_STATIC_ALLOCATION == 1 )
5      {
6          // 采用静态内存创建空闲任务
7          StaticTask_t *pxIdleTaskTCBBuffer = NULL;
8          StackType_t *pxIdleTaskStackBuffer = NULL;
9          uint32_t ulIdleTaskStackSize;
10         // 获取静态内存地址/参数
11         vApplicationGetIdleTaskMemory(
12             &pxIdleTaskTCBBuffer,
13             &pxIdleTaskStackBuffer,
14             &ulIdleTaskStackSize );
15         // 创建任务
16         // 空闲任务优先级为 0， 也就是其优先级最低
17         // ! ! 但是， 设置了特权位， 所以其运行在 特权模式
18         xIdleTaskHandle = xTaskCreateStatic(prvIdleTask, "IDLE",
19             ulIdleTaskStackSize, (void *) NULL,
20             (tskIDLE_PRIORITY | portPRIVILEGE_BIT),
21             pxIdleTaskStackBuffer,
22             pxIdleTaskTCBBuffer);
23
24         if( xIdleTaskHandle != NULL )
25         {
26             xReturn = pdPASS;
27         }
28         else
29         {
30             xReturn = pdFAIL;
31         }
32     }
33     #else
34     {
35         // 动态申请内存创建任务
36         xReturn = xTaskCreate(prvIdleTask,
37             "IDLE", configMINIMAL_STACK_SIZE,
38             (void *)NULL,
39             (tskIDLE_PRIORITY | portPRIVILEGE_BIT),
40             &xIdleTaskHandle );
41     }
42     #endif
43
44     // 如果工程使用了软件定时器， 需要创建定时器任务进行管理
45     #if ( configUSE_TIMERS == 1 )
46     {
47         if( xReturn == pdPASS )
48         {
49             xReturn = xTimerCreateTimerTask();
```

```
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    #endif

    if( xReturn == pdPASS )
    {

        // 关闭中断， 避免调度器运行前节拍定时器产生中断
        // 中断在第一个任务启动时恢复
        portDISABLE_INTERRUPTS();

        #if ( configUSE_NEWLIB_REENTRANT == 1 )
        {
            // 如果使用了这个库
            // 更新第一个任务的的指针到全局变量
            _impure_ptr = &( pxCurrentTCB->xNewLib_reent );
        }
        #endif

        // 初始化变量
        xNextTaskUnblockTime = portMAX_DELAY;
        xSchedulerRunning = pdTRUE;
        xTickCount = ( TickType_t ) 0U;

        // 如果启动统计任务运行时间， 宏 configGENERATE_RUN_TIME_STATS = 1
        // 需要定义以下宏， 初始化一个定时器用于该功能
        portCONFIGURE_TIMER_FOR_RUN_TIME_STATS();

        // 设置系统节拍计数器， 启动任务
        // 硬件相关， 由系统移植层提供， 下面介绍
        if( xPortStartScheduler() != pdFALSE )
        {
            // 不会运行到这里， 如果调度器运行正常
        }
        else
        {
            // 当调用 xTaskEndScheduler()才会来到这里
        }
    }
    else
    {
        // 内存不足，创建空闲任务/定时任务失败， 调度器启动失败
        configASSERT( xReturn != errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY );
    }

    // 预防编译器警告
    ( void ) xIdleTaskHandle;
}
```

移植层调度器

上面提到， 创建系统所需任务和初始化相关静态变量后， 系统调用了 `xPortStartScheduler` 设置节拍定时器和启动第一个任务，开始系统正常运行调度。 而对于不同数的实现可能存在不同，以下， 拿比较常用的 Cotex-M3 架构举例。
对于 M3， 可以在源码目录下 /Source/portable/GCC/ARM_CM3/port.c 看到该函数的实现。

与 FreeRTOS 任务优先级相反， Cotex-M3 优先级值越小， 优先级越高。 Cotex-M3的优先级配置寄存器考虑器件移植而向高位对齐，实际可用的 CPU 会裁掉表达优先级低少优先级数。 举例子说， 加入平台支持3bit 表示优先级，则其优先级配置寄存器的高三位可以编程写入，其他位被屏蔽，不管写入何值，重新读回都是0。
另外提供抢占优先级和子优先级分段配置相关，详细阅读《Cortex-M3权威指南》

在系统调度过程中，主要涉及到的三个异常：

SVC 系统服务调用
操作系统通常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。 这里主要触发后，在异常服务中启动第一个任务

PendSV 可悬起系统调用
相比 SVC， PenndSV 异常后可能不会马上响应， 等到其他高优先级中断处理后才响应。 用于上下文切换，同时保证其他中断可以被及时响应处理。

SysTick 节拍定时器
在没有高优先级任务强制下，同优先级任务按时间片轮流执行，每次SysTick中断，下一个任务将获得一个时间片。

```c
BaseType_t xPortStartScheduler( void )
{
    configASSERT( configMAX_SYSCALL_INTERRUPT_PRIORITY );
    #if( configASSERT_DEFINED == 1 )
    {
        volatile uint32_t ulOriginalPriority;
        // 取出中断优先级寄存器
        volatile uint8_t * const pucFirstUserPriorityRegister =
            (volatile uint8_t * const) (portNVIC_IP_REGISTERS_OFFSET_16 +
                        portFIRST_USER_INTERRUPT_NUMBER);
        volatile uint8_t ucMaxPriorityValue;

        // 保存原有优先级寄存器值
        ulOriginalPriority = *pucFirstUserPriorityRegister;

        // 判断平台支持优先级位数
        // 先全写 1
        *pucFirstUserPriorityRegister = portMAX_8_BIT_VALUE;
        // 重新读回， 不能设置的位依然是 0
        ucMaxPriorityValue = *pucFirstUserPriorityRegister;
        // 确保用户设置优先级不会超出范围
        ucMaxSysCallPriority = configMAX_SYSCALL_INTERRUPT_PRIORITY & ucMaxPriorityValue;

        // 判断有几个1， 得到对应优先级数最大值
        ulMaxPRIGROUPValue = portMAX_PRIGROUP_BITS;
        while( ( ucMaxPriorityValue & portTOP_BIT_OF_BYTE ) == portTOP_BIT_OF_BYTE )
        {
            ulMaxPRIGROUPValue--;
            ucMaxPriorityValue <<= ( uint8_t ) 0x01;
        }
        ulMaxPRIGROUPValue <<= portPRIGROUP_SHIFT;
        ulMaxPRIGROUPValue &= portPRIORITY_GROUP_MASK;

        // 恢复优先级配置寄存器值
        *pucFirstUserPriorityRegister = ulOriginalPriority;
    }
    #endif /* conifgASSERT_DEFINED */

    // 设置 PendSV 和 SysTIck 异常优先级最低
    // 保证系统会话切换不会阻塞系统其他中断的响应
    portNVIC_SYSPRI2_REG |= portNVIC_PENDSV_PRI;
    portNVIC_SYSPRI2_REG |= portNVIC_SYSTICK_PRI;

    // 初始化系统节拍定时器
    vPortSetupTimerInterrupt();
    // 初始化边界嵌套计数器
    uxCriticalNesting = 0;

    // 触发 svc 异常 启动第一个任务
    prvPortStartFirstTask();

    /* Should not get here! */
    prvTaskExitError();
    return 0;
}
```

启动第一个任务

函数中调用了 `prvPortStartFirstTask` 来启动第一个任务， 该函数重新初始化了系统的栈指针，表示 FreeRtos 开始接手平台的控制， 同时通过触发 SVC 系统调用务。具体实现如下

```c
static void prvPortStartFirstTask( void )
{
    __asm volatile(
    " ldr r0, =0xE000ED08   \n" /*向量表偏移寄存器地址 CotexM3*/
    " ldr r0, [r0]          \n" /*取向量表地址*/
    " ldr r0, [r0]          \n" /*取 MSP 初始值*/
    /*重置msp指针 宣示 系统接管*/
    " msr msp, r0           \n"
    " cpsie i               \n" /*开中断*/
    " cpsie f               \n" /*开异常*/
    /*流水线相关*/
    " dsb                   \n" /*数据同步隔离*/
```

```
    " isb                   \n" /*指令同步隔离*/
    /*触发异常 启动第一个任务*/
    " svc 0                 \n"
    " nop                   \n"
    );
}
```

前面创建任务的文章介绍过， 任务创建后， 对其栈进行了初始化，使其看起来和任务运行过后被系统中断切换了一样。 所以，为了启动第一个任务，触发 SVC 异常后，异常执行现场恢复， 把 `pxCurrentTCB` "恢复"到运行状态。

（另外，Cotex-M3 具有三级流水线，所以切换任务的时候需要清除预取的指令，避免错误。）

对于 Cotex-M3 ， 其代码实现如下，

```
void vPortSVCHandler( void )
{
    __asm volatile (
    /*取 pxCurrentTCB 的地址*/
    "ldr r3, pxCurrentTCBConst2      \n"
    /*取出 pxCurrentTCB 的值 ： TCB 地址*/
    "ldr r1, [r3]                    \n"
    /*取出 TCB 第一项 ： 任务的栈顶 */
    "ldr r0, [r1]                   \n"
    /*恢复寄存器数据*/
    "ldmia r0!, {r4-r11}            \n"
    /*设置线程指针： 任务的栈指针*/
    "msr psp, r0                    \n"
    /*流水线清洗*/
    "isb                            \n"
    "mov r0, #0                     \n"
    "msr    basepri, r0             \n"
    /*设置返回后进入线程模式*/
    "orr r14, #0xd                  \n"
    "bx r14                         \n"
    "                               \n"
    ".align 4               \n"
    "pxCurrentTCBConst2: .word pxCurrentTCB     \n"
    );
}
```

异常返回后， 系统进入线程模式， 自动从堆栈恢复PC等寄存器，而由于此时栈指针已经更新指向对应准备运行任务的栈，所以，程序会从该任务入口函数开始执行。
到此， 第一个任务启动。

前面提到， 第一个任务启动通过 SVC 异常， 而后续的任务切换， 使用的是 PendSV 异常， 而其对应的服务函数是 `xPortPendSVHandler` 。 后续介绍任务切换再分析

任务切换

FreeRTOS 支持时间片轮序和优先级抢占。系统调度器通过调度算法确定当前需要获得CPU 使用权的任务并让其处于运行状态。对于嵌入式系统，某些任务需要获得快速的响间片，该任务可能无法及时被运行，因此抢占调度是必须的，高优先级的任务一旦就绪就能及时运行；而对于同优先级任务，系统根据时间片调度，给予每个任务相同的运行时任务都能获得CPU 。

1. 最高优先级任务 Task 1 运行，直到其被阻塞或者挂起释放CPU
2. 就绪链表中最高优先级任务Task 2 开始运行， 直到...
   i. 调用接口进入阻塞或者挂起状态
   ii. 任务 Task 1 恢复并抢占 CPU 使用权
   iii. 同优先级任务TASK 3 就绪，时间片调度
3. 没有用户任务执行，运行系统空闲任务。

FreeRTOS 在两种情况下执行任务切换：

1. 同等级任务时间片用完，提前挂起触发切换
   在 SysTick 节拍计数器中断中触发异常
2. 高优先任务恢复就绪（如信号量，队列等阻塞、挂起状态下退出）时抢占
   最终都是通过调用移植层提供的 `portYIELD()` 宏悬起 PendSV 异常

但是无论何种情况下，都是通过触发系统 PendSV 异常，在该服务程序中完成切换。
使用该异常切换上下文的原因是保证切换不会影响到其他中断的及时响应（切换上下文抢占了 ISR 的执行，延时时间不可预知，对于实时系统是无法容忍的），在SysTick 中务切换的地方悬起一个 PendSV 异常，系统会直到其他所有 ISR 都完成处理后才执行该异常的服务程序，进行上下文切换。

系统响应 PendSV 异常，在该中断服务程序中，保存当前任务现场， 选择切换的下一个任务，进行任务切换，退出异常恢复线程模式运行新任务，完成任务切换。

以下是 Cotex-M3 的服务程序，
首先先要明确的是，系统进入异常处理程序的时候，使用的是主堆栈指针 MSP， 而一般情况下运行任务使用的线程模式使用的是进程堆栈指针 PSP。后者使用是系统设置的设置的。

对应这两个指针，系统有两种堆栈，系统内核和异常程序处理使用的是主堆栈，MSP 指向其栈顶。而对应而不同任务，我们在创建时为其分配了空间，作为该任务的堆栈，在由系统设置进程堆栈 PSP 指向该栈顶。

如下分析该服务函数的执行：

```
void xPortPendSVHandler( void )
{
    /* This is a naked function. */
    __asm volatile
    (
    /*取出当前任务的栈顶指针 也就是 psp -> R0*/
    "   mrs r0, psp                         \n"
    "   isb                                 \n"
    "                                       \n"
    /*取出当前任务控制块指针 -> R2*/
    "   ldr r3, pxCurrentTCBConst           \n"
    "   ldr r2, [r3]                        \n"
    "                                       \n"
    /*R4-R11 这些系统不会自动入栈，需要手动推到当前任务的堆栈*/
    "   stmdb r0!, {r4-r11}                 \n"
    /*最后，保存当前的栈顶指针
    R0 保存当前任务栈顶地址
    [R2] 是 TCB 首地址，也就是 pxTopOfStack
    下次，任务激活可以重新取出恢复栈顶，并取出其他数据
    */
    "   str r0, [r2]                        \n"
    "                                       \n"
    /*保护现场，调用函数更新下一个准备运行的新任务*/
    "   stmdb sp!, {r3, r14}                \n"
    /*设置优先级 第一个参数，
    即:configMAX_SYSCALL_INTERRUPT_PRIORITY
    进入临界区*/
    "   mov r0, %0                          \n"
    "   msr basepri, r0                     \n"
    "   bl vTaskSwitchContext               \n"
    "   mov r0, #0                          \n"
    "   msr basepri, r0                     \n"
    "   ldmia sp!, {r3, r14}                \n"
    "                                       \n"
    /*函数返回 退出临界区
    pxCurrentTCB 指向新任务
    取出新的 pxCurrentTCB 保存到 R1
    */
    "   ldr r1, [r3]                        \n"
    /*取出新任务的栈顶*/
    "   ldr r0, [r1]                        \n"
    /*恢复手动保存的寄存器*/
    "   ldmia r0!, {r4-r11}                 \n"
    /*设置线程指针 psp 指向新任务栈顶*/
    "   msr psp, r0                         \n"
    "   isb                                 \n"
    /*返回， 硬件执行现场恢复
    开始执行任务
    */
    "   bx r14                              \n"
    "                                       \n"
    "   .align 4                            \n"
    "pxCurrentTCBConst: .word pxCurrentTCB  \n"
    ::"i"(configMAX_SYSCALL_INTERRUPT_PRIORITY)
    );
}
```

在服务程序中，调用了函数 `vTaskSwitchContext` 获取新的运行任务， 该函数会更新当前任务运行时间，检查任务堆栈使用是是否溢出，然后调用宏 `taskSELECT_HIGHEST_PRIORITY_TASK()` 设置新的任务。该宏实现分两种情况，普通情况下使用的定义如下

```
UBaseType_t uxTopPriority = uxTopReadyPriority;
while(listLIST_IS_EMPTY(&(pxReadyTasksLists[uxTopPriority])))
{
    --uxTopPriority;
}

listGET_OWNER_OF_NEXT_ENTRY(pxCurrentTCB,
    &(pxReadyTasksLists[ uxTopPriority]));

```

```
10
    uxTopReadyPriority = uxTopPriority;
```

通过 while 查找当前存在就绪任务的最高优先级链表，获取链表项设置任务指针。（通一个链表内多个项目通过指针循环，实现同优先级任务获得相同时间片执行）。

而另外一种方式，需要平台支持，主要差别是查找最高任务优先级，平台支持利用平台特性，效率会更高，但是移植性就不好说了。

发生异常跳转到异常处理服务前，自动执行的现场保护会保留返回模式（线程模式），使用堆栈指针等信息，所以，结束任务切换， 通过执行 `bx r14` 返回，系统会自
（From stack），开始运行任务。

至此，任务切换完成。

参考

source code

《Cortex-M3权威指南》