title: 【TencentOS tiny】深度源码分析（2）——调度器 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-12 23:10:54 img: coverImg: password: summary: tags: - TencentOS tiny - RTOS - 操作系统 - 物联网 categories: - 操作系统 - TencentOS tiny

> 温馨提示：本文不描述与浮点相关的寄存器的内容，如需了解自行查阅（毕竟我自己也不懂）

# 调度器的基本概念

TencentOS tiny中提供的任务调度器是基于优先级的全抢占式调度，在系统运行过程中，当有比当前任务优先级更高的任务就绪时，当前任务将立刻被切出，高优先级任务抢占处理器运行。

TencentOS tiny内核中也允许创建相同优先级的任务。相同优先级的任务采用时间片轮转方式进行调度（也就是通常说的分时调度器），时间片轮转调度仅在当前系统中**无更高优先级就绪任务**的情况下才有效。

为了保证系统的实时性，系统尽最大可能地保证高优先级的任务得以运行。任务调度的原则是一旦任务状态发生了改变，并且当前运行的任务优先级小于优先级队列中任务最高优先级时，立刻进行任务切换（除非当前系统处于中断处理程序中或禁止任务切换的状态）。

调度器是操作系统的核心，其主要功能就是实现任务的切换，即从就绪列表里面找到优先级最高的任务，然后去执行该任务。

# 启动调度器

调度器的启动由cpu_sched_start函数来完成，它会被tos_knl_start函数调用，这个函数中主要做两件事，首先通过readyqueue_highest_ready_task_get函数获取当前系统中处于最高优先级的就绪任务，并且将它赋值给指向当前任务控制块的指针k_curr_task ，然后设置一下系统的状态为运行态KNL_STATE_RUNNING。

当然最重要的是调用汇编代码写的函数cpu_sched_start启动调度器，该函数在源码的arch\arm\arm-v7m目录下的port_s.S汇编文件下，TencentOS tiny支持多种内核的芯片，如M3/M4/M7等，不同的芯片该函数的实现方式不同，port_s.S也是TencentOS tiny作为软件与CPU硬件连接的桥梁。以M4的cpu_sched_start举个例子：

```
__API__ k_err_t tos_knl_start(void)
{
    if (tos_knl_is_running()) {
        return K_ERR_KNL_RUNNING;
    }

    k_next_task = readyqueue_highest_ready_task_get();
    k_curr_task = k_next_task;
    k_knl_state = KNL_STATE_RUNNING;
    cpu_sched_start();

    return K_ERR_NONE;
}
```

```
port_sched_start
    CPSID   I

    ; set pendsv priority lowest
    ; otherwise trigger pendsv in port_irq_context_switch will cause a context
swich in irq
    ; that would be a disaster
    MOV32   R0, NVIC_SYSPRI14
    MOV32   R1, NVIC_PENDSV_PRI
    STRB    R1, [R0]

    LDR     R0, =SCB_VTOR
    LDR     R0, [R0]
    LDR     R0, [R0]
    MSR     MSP, R0

    ; k_curr_task = k_next_task
    MOV32   R0, k_curr_task
    MOV32   R1, k_next_task
    LDR     R2, [R1]
    STR     R2, [R0]

    ; sp = k_next_task->sp
    LDR     R0, [R2]
    ; PSP = sp
    MSR     PSP, R0

    ; using PSP
    MRS     R0, CONTROL
    ORR     R0, R0, #2
    MSR     CONTROL, R0

    ISB

    ; restore r4-11 from new process stack
    LDMFD   SP!, {R4 - R11}

    IF {FPU} != "SoftVFP"
    ; ignore EXC_RETURN the first switch
    LDMFD   SP!, {R0}
    ENDIF

    ; restore r0, r3
    LDMFD    SP!, {R0 - R3}
    ; load R12 and LR
    LDMFD    SP!, {R12, LR}
    ; load PC and discard xPSR
    LDMFD    SP!, {R1, R2}

    CPSIE    I
    BX       R1
```

## Cortex-M内核关中断指令

从上面的汇编代码，我又想介绍一下Cortex-M内核关中断指令，唉~感觉还是有点麻烦！ 为了快速地开关中断， Cortex-M内核专门设置了一条 CPS 指令，用于操作PRIMASK寄存器跟FAULTMASK寄存器的，这两个寄存器是与屏蔽中断有关的，除此之外Cortex-M内核还存在BASEPRI寄存器也是与中断有关的，也顺带介绍一下吧。

```
CPSID I         ;PRIMASK=1      ;关中断
CPSIE I         ;PRIMASK=0      ;开中断
CPSID F         ;FAULTMASK=1    ;关异常
CPSIE F         ;FAULTMASK=0    ;开异常
```

| 寄存器 | 功能 |
| --- | --- |
| PRIMASK | 它被置 1 后，就关掉所有可屏蔽的异常，只剩下 NMI 和HardFault FAULT可以响应 |
| FAULTMASK | 当它置 1 时，只有 NMI 才能响应，所有其它的异常都无法响应（包括HardFault FAULT） |
| BASEPRI | 这个寄存器最多有 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断 |

更多具体的描述看我以前的文章：RTOS临界段知识：https://blog.csdn.net/jiejiemcu/article/details/82534974

## 回归正题

在启动内核调度器过程中需要配置PendSV 的中断优先级为最低，就是往NVIC_SYSPRI14（0xE000ED22）地址写入NVIC_PENDSV_PRI（0xFF）。因为PendSV都会涉及到系统调度，系统调度的优先级要低于系统的其它硬件中断优先级，即优先响应系统中的外部硬件中断，所以PendSV的中断优先级要配置为最低，不然很可能在中断上下文中产生任务调度。

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 ISR 正在活动，它将悬起一个 PendSV 异常，以便缓期执行上下文切换。也就是说，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ISR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：

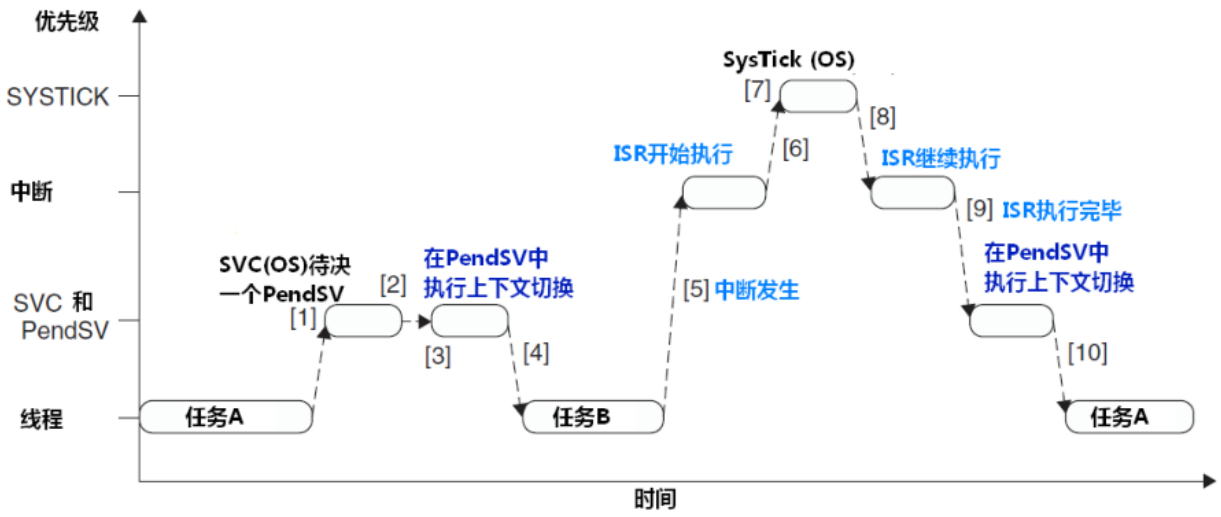


图 7.17 使用 PendSV 控制上下文切换

然后获取MSP主栈指针的地址，在Cortex-M中，0xE000ED08是SCB_VTOR寄存器的地址，里面存放的是向量表的起始地址。

加载k_next_task指向的任务控制块到 R2，从上一篇文章可知任务控制块的第一个成员就是栈顶指针，所以此时R2等于栈顶指针。

> ps：在调度器启动时，k_next_task与k_curr_task是一样的（k_curr_task = k_next_task）

加载R2到R0，然后将栈顶指针R0更新到psp，任务执行的时候使用的栈指针是psp。

> ps：sp指针有两个，分别为psp和msp。（可以简单理解为：在任务上下文环境中使用psp，在中断上下文环境使用msp，也不一定是正确的，这是我个人的理解）

以R0为基地址，将栈中向上增长的8个字的内容加载到CPU寄存器R4~R11，同时R0也会跟着自增

接着需要加载R0 ~ R3、R12以及LR、 PC、xPSR到CPU寄存器组，PC指针指向的是即将要运行的线程，而LR寄存器则指向任务的退出。因为这是第一次启动任务，要全部手动把任务栈上的寄存器弹到硬件里，才能进入第一个任务的上下文，因为一开始并没有第一个任务运行的上下文环境，而在进入PendSV的时候需要上文保存，所以需要手动创造任务上下文环境（将这些寄存器加载到CPU寄存器组中），第一次的时候此汇编入口函数，sp是指向一个选好的任务的栈顶（k_curr_task）。

## 看看任务栈的初始化

从上面的了解，再来看看任务栈的初始化，可能会有更深一点的印象。主要了解以下几点即可：

- 获取栈顶指针为stk_base[stk_size]高地址，Cortex-M内核的栈是向下增长的。
- R0、R1、R2、R3、R12、R14、R15和xPSR的位24是会被CPU自动加载与保存的。
- xPSR的bit24必须置1，即0x01000000。
- entry是任务的入口地址，即PC
- R14 (LR)是任务的退出地址，所以任务一般是死循环而不会return
- R0: arg是任务主体的形参
- 初始化栈时sp指针会自减

```
__KERNEL__ k_stack_t *cpu_task_stk_init(void *entry,
                                        void *arg,
```

```
                                                void *exit,
                                                k_stack_t *stk_base,
                                                size_t stk_size)
{
    cpu_data_t *sp;

    sp = (cpu_data_t *)&stk_base[stk_size];
    sp = (cpu_data_t *)((cpu_addr_t)(sp) & 0xFFFFFFF8);

    /* auto-saved on exception(pendSV) by hardware */
    *--sp = (cpu_data_t)0x01000000u;    /* xPSR     */
    *--sp = (cpu_data_t)entry;          /* entry    */
    *--sp = (cpu_data_t)exit;           /* R14 (LR) */
    *--sp = (cpu_data_t)0x12121212u;    /* R12      */
    *--sp = (cpu_data_t)0x03030303u;    /* R3       */
    *--sp = (cpu_data_t)0x02020202u;    /* R2       */
    *--sp = (cpu_data_t)0x01010101u;    /* R1       */
    *--sp = (cpu_data_t)arg;            /* R0: arg  */

    /* Remaining registers saved on process stack */
    /* EXC_RETURN = 0xFFFFFFFDL
       Initial state: Thread mode +  non-floating-point state + PSP
       31 - 28 : EXC_RETURN flag, 0xF
       27 -  5 : reserved, 0xFFFFFE
       4       : 1, basic stack frame; 0, extended stack frame
       3       : 1, return to Thread mode; 0, return to Handler mode
       2       : 1, return to PSP; 0, return to MSP
       1       : reserved, 0
       0       : reserved, 1
     */
#if defined (TOS_CFG_CPU_ARM_FPU_EN) && (TOS_CFG_CPU_ARM_FPU_EN == 1U)
    *--sp = (cpu_data_t)0xFFFFFFFDL;
#endif

    *--sp = (cpu_data_t)0x11111111u;    /* R11      */
    *--sp = (cpu_data_t)0x10101010u;    /* R10      */
    *--sp = (cpu_data_t)0x09090909u;    /* R9       */
    *--sp = (cpu_data_t)0x08080808u;    /* R8       */
    *--sp = (cpu_data_t)0x07070707u;    /* R7       */
    *--sp = (cpu_data_t)0x06060606u;    /* R6       */
    *--sp = (cpu_data_t)0x05050505u;    /* R5       */
    *--sp = (cpu_data_t)0x04040404u;    /* R4       */

    return (k_stack_t *)sp;
}
```

# 查找最高优先级任务

一个操作系统如果只是具备了高优先级任务能够立即获得处理器并得到执行的特点，那么它仍然不算是实时操作系统。因为这个查找最高优先级任务的过程决定了调度时间是否具有确定性，可以简单来说可以使用时间复杂度来描述一下吧，如果系统查找最高优先级任务的时间是O(N)，那么这个时间会随着任务个数的增加而增

大，这是不可取的，TencentOS tiny的时间复杂度是O(1)，它提供两种方法查找最高优先级任务，通过TOS_CFG_CPU_LEAD_ZEROS_ASM_PRESENT宏定义决定。

1. 第一种是使用普通方法，根据就绪列表中k_rdyq.prio_mask[]的变量判断对应的位是否被置1。
2. 第二种方法则是特殊方法，利用计算前导零指令CLZ，直接在k_rdyq.prio_mask[]这个32位的变量中直接得出最高优先级所处的位置，这种方法比普通方法更快捷，但受限于平台（需要硬件前导零指令，在STM32中我们就可以使用这种方法）。

实现过程如下，建议看一看readyqueue_prio_highest_get函数，他的实现还是非常精妙的~

```
__STATIC__ k_prio_t readyqueue_prio_highest_get(void)
{
    uint32_t *tbl;
    k_prio_t prio;

    prio    = 0;
    tbl     = &k_rdyq.prio_mask[0];

    while (*tbl == 0) {
        prio += K_PRIO_TBL_SLOT_SIZE;
        ++tbl;
    }
    prio += tos_cpu_clz(*tbl);
    return prio;
}
```

```
__API__ uint32_t tos_cpu_clz(uint32_t val)
{
#if defined(TOS_CFG_CPU_LEAD_ZEROS_ASM_PRESENT) &&
(TOS_CFG_CPU_LEAD_ZEROS_ASM_PRESENT == 0u)
    uint32_t nbr_lead_zeros = 0;

    if (!(val & 0XFFFF0000)) {
        val <<= 16;
        nbr_lead_zeros += 16;
    }

    if (!(val & 0XFF000000)) {
        val <<= 8;
        nbr_lead_zeros += 8;
    }

    if (!(val & 0XF0000000)) {
        val <<= 4;
        nbr_lead_zeros += 4;
    }

    if (!(val & 0XC0000000)) {
        val <<= 2;
        nbr_lead_zeros += 2;
```

```
    }

    if (!(val & 0X80000000)) {
        nbr_lead_zeros += 1;
    }

    if (!val) {
        nbr_lead_zeros += 1;
    }

    return (nbr_lead_zeros);
#else
    return port_clz(val);
#endif
}
```

# 任务切换的实现

从前面我们也知道，任务切换是在PendSV中断中进行的，这个中断中实现的内容总结成一句精髓的话就是 上文保存，下文切换，直接看源代码：

```
PendSV_Handler
    CPSID   I
    MRS     R0, PSP

_context_save
    ; R0-R3, R12, LR, PC, xPSR is saved automatically here
    IF {FPU} != "SoftVFP"
    ; is it extended frame?
    TST     LR, #0x10
    IT      EQ
    VSTMDBEQ  R0!, {S16 - S31}
    ; S0 - S16, FPSCR saved automatically here

    ; save EXC_RETURN
    STMFD   R0!, {LR}
    ENDIF

    ; save remaining regs r4-11 on process stack
    STMFD   R0!, {R4 - R11}

    ; k_curr_task->sp = PSP
    MOV32   R5, k_curr_task
    LDR     R6, [R5]
    ; R0 is SP of process being switched out
    STR     R0, [R6]

_context_restore
    ; k_curr_task = k_next_task
    MOV32   R1, k_next_task
```

```
    LDR     R2, [R1]
    STR     R2, [R5]

    ; R0 = k_next_task->sp
    LDR     R0, [R2]

    ; restore R4 - R11
    LDMFD   R0!, {R4 - R11}

    IF {FPU} != "SoftVFP"
    ; restore EXC_RETURN
    LDMFD   R0!, {LR}
    ; is it extended frame?
    TST     LR, #0x10
    IT      EQ
    VLDMIAEQ    R0!, {S16 - S31}
    ENDIF

    ; Load PSP with new process SP
    MSR     PSP, R0
    CPSIE   I
    ; R0-R3, R12, LR, PC, xPSR restored automatically here
    ; S0 - S16, FPSCR restored automatically here if FPCA = 1
    BX      LR

    ALIGN
    END
```

将PSP的值存储到R0。当进入PendSVC_Handler时，上一个任务运行的环境即：xPSR，PC（任务入口地址），R14，R12，R3，R2，R1，R0这些CPU寄存器的值会自动存储到任务的栈中，此时psp指针已经被自动更新。而剩下的r4~r11需要手动保存，这也是为啥要在PendSVC_Handler中保存上文（_context_save）的原因，主要是加载CPU中不能自动保存的寄存器，将其压入任务栈中。

接着找到下一个要运行的任务k_next_task，将它的任务栈顶加载到R0，然后手动将新任务栈中的内容（此处是指R4~R11）加载到CPU寄存器组中，这就是下文切换，当然还有一些其他没法自动保存的内容也是需要手动加载到CPU寄存器组的。手动加载完后，此时R0已经被更新了，更新psp的值，在退出PendSVC_Handler中断时，会以psp作为基地址，将任务栈中剩下的内容（xPSR，PC（任务入口地址），R14，R12，R3，R2，R1，R0）自动加载到CPU寄存器。

其实在异常发生时，R14中保存异常返回标志，包括返回后进入任务模式还是处理器模式、使用PSP堆栈指针还是MSP堆栈指针。此时的r14等于0xfffffffd，最表示异常返回后进入任务模式（毕竟PendSVC_Handler优先级是最低的，会返回到任务中），SP以PSP作为堆栈指针出栈，出栈完毕后PSP指向任务栈的栈顶。当调用 BX R14 指令后，系统以PSP作为SP指针出栈，把接下来要运行的新任务的任务栈中剩下的内容加载到CPU寄存器：R0、R1、R2、R3、R12、R14（LR）、R15（PC）和xPSR，从而切换到新的任务。

# SysTick

## SysTick初始化

systick是系统的时基，而且它是内核时钟，只要是M0/M3/M4/M7内核它都会存在systick时钟，并且它是可以被编程配置的，这就对操作系统的移植提供极大的方便。 TencentOS tiny会在cpu_init函数中将systick进行初始化，即调用cpu_systick_init函数，这样子就不需要用户自行去编写systick初始化相关的代码。

```
__KERNEL__ void cpu_init(void)
{
    k_cpu_cycle_per_tick = TOS_CFG_CPU_CLOCK / k_cpu_tick_per_second;
    cpu_systick_init(k_cpu_cycle_per_tick);

#if (TOS_CFG_CPU_HRTIMER_EN > 0)
    tos_cpu_hrtimer_init();
#endif
}
```

```
__KERNEL__ void cpu_systick_init(k_cycle_t cycle_per_tick)
{
    port_systick_priority_set(TOS_CFG_CPU_SYSTICK_PRIO);
    port_systick_config(cycle_per_tick);
}
```

## SysTick中断

SysTick中断服务函数是需要我们自己编写的，要在里面调用一下TencentOS tiny相关的函数，更新系统时基以驱动系统的运行，SysTick_Handler函数的移植如下：

```
void SysTick_Handler(void)
{
  HAL_IncTick();
  if (tos_knl_is_running())
  {
    tos_knl_irq_enter();

    tos_tick_handler();

    tos_knl_irq_leave();
  }
}
```

主要是需要调用tos_tick_handler函数将系统时基更新，具体见：

```
__API__ void tos_tick_handler(void)
{
    if (unlikely(!tos_knl_is_running())) {
        return;
    }
```

```
    tick_update((k_tick_t)1u);

#if TOS_CFG_TIMER_EN > 0u && TOS_CFG_TIMER_AS_PROC > 0u
    timer_update();
#endif

#if TOS_CFG_ROUND_ROBIN_EN > 0u
    robin_sched(k_curr_task->prio);
#endif
}
```

不得不说TencentOS tiny源码的实现非常简单，我非常喜欢，在tos_tick_handler中，首先判断一下系统是否已经开始运行，如果没有运行将直接返回，如果已经运行了，那就调用tick_update函数更新系统时基，如果使能了TOS_CFG_TIMER_EN 宏定义表示使用软件定时器，则需要更新相应的处理，此处暂且不提及。如果使能了TOS_CFG_ROUND_ROBIN_EN 宏定义，还需要更新时间片相关变量，稍后讲解。

```
__KERNEL__ void tick_update(k_tick_t tick)
{
    TOS_CPU_CPSR_ALLOC();
    k_task_t *first, *task;
    k_list_t *curr, *next;

    TOS_CPU_INT_DISABLE();
    k_tick_count += tick;

    if (tos_list_empty(&k_tick_list)) {
        TOS_CPU_INT_ENABLE();
        return;
    }

    first = TOS_LIST_FIRST_ENTRY(&k_tick_list, k_task_t, tick_list);
    if (first->tick_expires <= tick) {
        first->tick_expires = (k_tick_t)0u;
    } else {
        first->tick_expires -= tick;
        TOS_CPU_INT_ENABLE();
        return;
    }

    TOS_LIST_FOR_EACH_SAFE(curr, next, &k_tick_list) {
        task = TOS_LIST_ENTRY(curr, k_task_t, tick_list);
        if (task->tick_expires > (k_tick_t)0u) {
            break;
        }

        // we are pending on something, but tick's up, no longer waitting
        pend_task_wakeup(task, PEND_STATE_TIMEOUT);
    }
```

```
    TOS_CPU_INT_ENABLE();
}
```

tick_update函数的主要功能就是将k_tick_count +1，并且判断一下时基列表k_tick_list（也可以成为延时列表吧）的任务是否超时，如果超时则唤醒该任务，否则就直接退出即可。关于时间片的调度也是非常简单，将任务的剩余时间片变量timeslice减一，然后当变量减到0时，将该变量进行重装载timeslice_reload，然后切换任务knl_sched()，其实现过程如下：

```
__KERNEL__ void robin_sched(k_prio_t prio)
{
    TOS_CPU_CPSR_ALLOC();
    k_task_t *task;

    if (k_robin_state != TOS_ROBIN_STATE_ENABLED) {
        return;
    }

    TOS_CPU_INT_DISABLE();

    task = readyqueue_first_task_get(prio);
    if (!task || knl_is_idle(task)) {
        TOS_CPU_INT_ENABLE();
        return;
    }

    if (readyqueue_is_prio_onlyone(prio)) {
        TOS_CPU_INT_ENABLE();
        return;
    }

    if (knl_is_sched_locked()) {
        TOS_CPU_INT_ENABLE();
        return;
    }

    if (task->timeslice > (k_timeslice_t)0u) {
        --task->timeslice;
    }

    if (task->timeslice > (k_timeslice_t)0u) {
        TOS_CPU_INT_ENABLE();
        return;
    }

    readyqueue_move_head_to_tail(k_curr_task->prio);

    task = readyqueue_first_task_get(prio);
    if (task->timeslice_reload == (k_timeslice_t)0u) {
        task->timeslice = k_robin_default_timeslice;
    } else {
        task->timeslice = task->timeslice_reload;
```

```
    }

    TOS_CPU_INT_ENABLE();
    knl_sched();
}
```

# 喜欢就关注我吧！


相关代码可以在公众号后台获取。