初学RT\_Thread线程调度

学习RT\_thread线程调度与管理需要准备的基础知识主要有：C语言（数组、指针、结构体、结构体指针、链表、堆栈）；汇编的一些知识。推荐参考的资料：RT\_Thread编程指南（这个手册里面有函数的介绍可以很快的了解函数的功能），ARM Cortex-M0权威指南，C语言程序设计（谭浩强）。

刚接触RT\_Thread我首先看的是RT\_Thread编程指南，以为了解了接口函数就可以知道程序是怎么运行的了，就看这个手册看呀看呀，看过全忘了，哈哈。好吧，发现这样看不行，就开始看程序了。。。看到程序一脸懵逼（以前没接触过操作系统方面的内容，不知道怎么下手）。冷静一下，看了看文件目录，看到Applications文件夹下的.c文件，窃喜。感觉要愉快的开始学习了，找到main函数，先是关系统中断，然后就是对使用芯片的初始化（使能系统时钟，配置串口之类的），定时器管理系统初始化，调度器初始化，用户程序初始化（这里面是用户添加线程的地方，创建需要的线程，启动线程），初始化定时器线程，初始化空闲线程（这一步很重要，保证任何时刻都有任务可以执行，另外空闲线程里面会进行一些特殊的操作，对于需要删除的线程是在空闲线程里删除的），启动调度器（在这里面找出第一个准备好的线程，并交给系统去切换到准备好的线程）。呃呃，程序到这里没有了，下面没有了，该怎么办，啊啊啊，崩溃了，自己学艺不精，不知道怎么看了，按照以往的经验不应该有大循环吗？到底怎么切换线程的，好吧，想不明白，按照编程指南，在rt\_application\_init()里面创建了串口输出，led灯闪烁，并下载到板子里，程序跑起来了，led在闪烁，串口在打印信息，，，，

回到创建线程开始看：

rt\_thread\_t rt\_thread\_create(const char \*name,

void (\*entry)(void \*parameter),

void \*parameter,

rt\_uint32\_t stack\_size,

rt\_uint8\_t priority,

rt\_uint32\_t tick)

一句很重要的话：一个线程要成为可执行的对象就必须由操作系统的内核来为它创建（初始化）一个线程句柄。调用这个函数时，系统会长动态堆内存中分配一个线程句柄（线程控制块）以及按照参数中指定的堆大小从动态堆中分配相应的空间。

函数参数

|  |  |
| --- | --- |
| 参数 | 描述 |
| name | 线程的名称；线程名称的最大长度由rtconfig.h中定义的RT\_NAME\_MAX宏定义，多余部分会被自动裁掉 |
|  |  |
| entry | 线程入口函数 |
| parameter | 线程入口函数参数 |
| stack\_size | 线程栈大小，单位是字节。在打多数系统中需要做栈空间地址对齐（例如ARM体系结构中需要向4字节地址对齐）。 |
| priority | 线程的优先级。线程优先级范围根据系统配置情况（rtcinfig.h中的RT\_THREAD\_PRIORITY\_MAX宏定义），如果支持的是256级优先级，那么范围是从0-255，数值越小优先级越高，0代表最高优先级。 |
| tick | 线程的时间片大小。时间片（tick）的单位是操作系统的时钟节拍。当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度。这个时间片运行结束时，调度器自动选择下一个就绪态的同优先级线程进行运行。 |

创建过线程之后判断有没有创建成功，如果创建成功就执行线程启动。

rt\_thread\_startup(rt\_thread\_t thread)

函数参数

|  |  |
| --- | --- |
| 参数 | 描述 |
| thread | 线程句柄 |

通过这个函数将线程设置为更改为就绪状态，并放到相应优先级队列中等待调度。注：虽然这个函数里面有调用调度函数，但是没有满足条件不会进行线程调度。（好吧我刚开始接触的时候并没有意识到这一点，一直以为在这就执行调度了，并且认为每次都是在这进行的调度）。

以上就是我刚开始接触RT\_Thread线程调度时走的弯路。分享给大家。下面介绍一下正确的调度学习过程。



图1 初始化流程图

可以看到初始化的流程都是套路，但是有些需要注意的点

1：配置SYSTick：SysTick：系统滴答定时器（也就是周期性溢出的时基定时器），用于在每隔一定的时间产生一个中断，即使系统在睡眠模式下也能工作。可以通过SYSTICK进行线程的调度

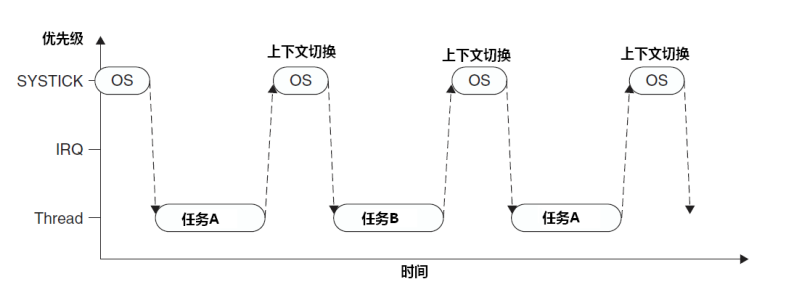


图2 两个任务间通过SYSTICK轮转调度的简单模式

上图是两个任务轮转调度的示意图。但若在产生systick异常时正在响应一个中断，则systick异常会抢占其ISR。在这种情况下，OS不得执行上下文切换，否则将使中断请求被延迟。如果OS在某中断活跃时尝试切入线程模式，将触犯用法fault异常。

需要说明的是，线程的上下文切换还是通过systick触发的，接下来会介绍另一个异常，来解决上面的问题。

创建线程的时候有一个参数是tick(线程的时间片大小)，通过这个参数设置这个线程单次运行的最长时间，达到这个时间时就会触发systick异常。然后systick异常会触发一个PendSV异常。在响应一个中断时PendSV异常会自动延迟上下切换的请求，直到其他的ISR都完成了处理后才进行切换。为实现这个机制，需要把PendSV编程为最低优先级的异常。

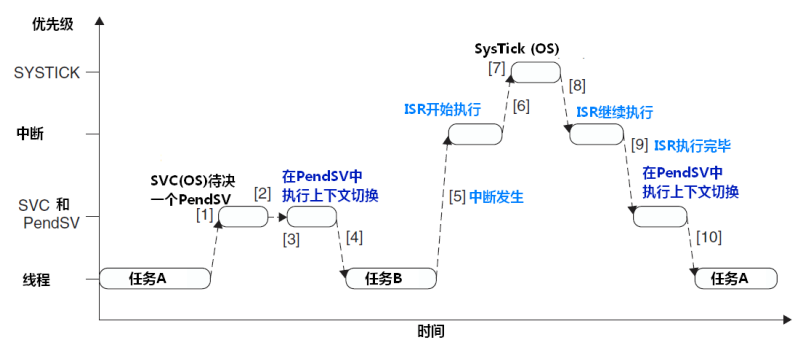


图3 使用PendSV控制上下文切换

/\*\*

\* This is the timer interrupt service routine.

\*

\*/

void SysTick\_Handler(void)

{

/\* enter interrupt \*/

rt\_interrupt\_enter();

rt\_tick\_increase();

/\* leave interrupt \*/

rt\_interrupt\_leave();

}

SysTick中断服务函数，重点看rt\_tick\_increase();可以看到该函数检查当前线程剩余时间是否为0，如果为0就执行rt\_thread\_yield();调用该函数后，当前线程首先把自己从它所在的就绪优先级线程队列中删除，然后把自己挂到这个优先级队列链表的尾部，然后激活调度器进行线程上下文切换（如果当前优先级只有这一个线程，则这个线程继续执行，不进行上下文切换动作）。接着会调用rt\_schedule(void)，系统会计算一次系统中就绪态的线程，如果存在比当前线程更高优先级的线程时，系统将切换到高优先级的线程去。

rt\_schedule(void)这个函数中包含了一个调度的核心算法：位图调度算法，使用这个算法保证快速的计算出准备好的线程。

然后进入下面的汇编程序段

;/\*

; \* void rt\_hw\_context\_switch(rt\_uint32 from, rt\_uint32 to);

; \* r0 --> from

; \* r1 --> to

; \*/

rt\_hw\_context\_switch\_interrupt

EXPORT rt\_hw\_context\_switch\_interrupt

rt\_hw\_context\_switch PROC

EXPORT rt\_hw\_context\_switch

; set rt\_thread\_switch\_interrupt\_flag to 1

LDR r2, =rt\_thread\_switch\_interrupt\_flag

LDR r3, [r2]

CMP r3, #1

BEQ \_reswitch

MOVS r3, #0x01

STR r3, [r2]

LDR r2, =rt\_interrupt\_from\_thread ; set rt\_interrupt\_from\_thread

STR r0, [r2]

\_reswitch

LDR r2, =rt\_interrupt\_to\_thread ; set rt\_interrupt\_to\_thread

STR r1, [r2]

LDR r0, =NVIC\_INT\_CTRL ; trigger the PendSV exception (causes context switch)

LDR r1, =NVIC\_PENDSVSET

STR r1, [r0]

BX LR

ENDP

; r0 --> switch from thread stack

; r1 --> switch to thread stack

; psr, pc, lr, r12, r3, r2, r1, r0 are pushed into [from] stack

PendSV\_Handler PROC

EXPORT PendSV\_Handler

; disable interrupt to protect context switch

MRS r2, PRIMASK ;¶ÁÈ¡PRIMASK£¬±£´æµ½r2ÖÐ

CPSID I ;¹ØÖÐ¶Ï

; get rt\_thread\_switch\_interrupt\_flag

LDR r0, =rt\_thread\_switch\_interrupt\_flag

LDR r1, [r0]

CMP r1, #0x00

BEQ pendsv\_exit ; pendsv already handled

; clear rt\_thread\_switch\_interrupt\_flag to 0

MOVS r1, #0x00

STR r1, [r0]

LDR r0, =rt\_interrupt\_from\_thread

LDR r1, [r0]

CMP r1, #0x00

BEQ switch\_to\_thread ; skip register save at the first time

MRS r1, psp ; get from thread stack pointer

SUBS r1, r1, #0x20 ; space for {r4 - r7} and {r8 - r11}

LDR r0, [r0]

STR r1, [r0] ; update from thread stack pointer

STMIA r1!, {r4 - r7} ; push thread {r4 - r7} register to thread stack

MOV r4, r8 ; mov thread {r8 - r11} to {r4 - r7}

MOV r5, r9

MOV r6, r10

MOV r7, r11

STMIA r1!, {r4 - r7} ; push thread {r8 - r11} high register to thread stack

switch\_to\_thread

LDR r1, =rt\_interrupt\_to\_thread

LDR r1, [r1]

LDR r1, [r1] ; load thread stack pointer

LDMIA r1!, {r4 - r7} ; pop thread {r4 - r7} register from thread stack

PUSH {r4 - r7} ; push {r4 - r7} to MSP for copy {r8 - r11}

LDMIA r1!, {r4 - r7} ; pop thread {r8 - r11} high register from thread stack to {r4 - r7}

MOV r8, r4 ; mov {r4 - r7} to {r8 - r11}

MOV r9, r5

MOV r10, r6

MOV r11, r7

POP {r4 - r7} ; pop {r4 - r7} from MSP

MSR psp, r1 ; update stack pointer

pendsv\_exit

; restore interrupt

MSR PRIMASK, r2

MOVS r0, #0x04

RSBS r0, r0, #0x00

BX r0

ENDP

通过这两段汇编程序实现线程的切换。