# 任务调度机制总结

## 任务调度器的总结

### 1、启动调度器

**void**vTaskStartScheduler( **void** ) —— （1）接口函数

（2）函数主要实现的功能

- Ⅰ根据宏支持，进行动态/静态空闲任务的创建（一般是动态）
- Ⅱ、sys计数的初始化和下一任务解锁时刻初始化
- Ⅲ、调用接口xPortStartScheduler完成优先级的配置（sys中断配置和系统最大优先级）
- Ⅳ、开启第一个任务（第一次呼叫pendSv）

### 2、任务调度的实现

核心函数——任务切换函数通过这个函数，就可以不断寻找最高优先级的任务去执行<taskYIELD（）>

任务调度是靠systick中断实现的

具体为：在systick中断当中更新时间，判断是否有任务需要解锁（解锁就更新优先级，更新pxcurrentTCB，以此判断是否需要进行任务切换），如果解锁的任务优先级更大就进行任务切换

任务切换：本质上是在不断的在SysTick中断当中呼叫PendSV中断，其内进行完成任务切换（具体见PendSV中断服务函数）==>也就是一退出sys中断以后，任务就完成了切换，就会执行另一个任务

SysTick相当于一个心脏，在一次次的SysTick计时到期后进入Sys中断，检查是否解锁新任务，更新任务优先级，呼叫PendSV完成任务切换

其实无论解锁与否，sys都必然会呼叫一个PendSV中断，在其内完成让任务切换：1）优先级如果改变，就会切换到任务就绪列表的另一优先级下进行执行

2）如果没变就会保持在当前优先级就绪列表下的任务当中执行

（支持时间片，同一优先级的任务列表下的任务以一个Tick为周期轮流执行）<原因见左图>

```
/* Tasks of equal priority to the currently running task will share
processing time (time slice) if preemption is on, and the application
writer has not explicitly turned time slicing off. */
#if ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING == 1 ) )
    {
        if( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[ pxCurrentTCB->uxPriority ] ) ) > ( UBaseType_t ) 1 )
        {
            xSwitchRequired = pdTRUE;
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
```

在前面的解锁判断完成后，如果支持抢占式任务调度和时间片，就必然会请求任务切换

### 3、多任务和时间片的实现——基于任务阻塞和延时列表

**在rtos当中的延时为vTaskDelay（），是一种阻塞延时，我们从两个视角来看待这种延时**

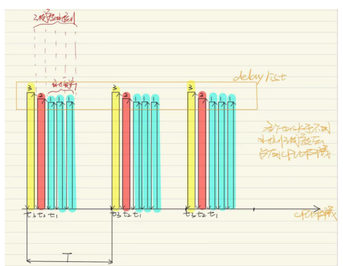
（1）**从当前任务本身来看：**由于任务是一个死循环，此时的阻塞*对于当前任务来说就相当于延时一段时间*，比如led0n，vtaskdelay，ledoff…对于这个任务而言，*其运行效果无异于使用delay*，实现了led的闪烁

（2）**但对于其他任务来说**：此处为阻塞延时，会把原任务添加到阻塞列表，随后进行任务切换，*此时给其他任务“腾出”机会，得以切换到其他任务去执行*

（3）**时间片**——比如我们定义了两个任务task1和task2（优先级相等），分别执行led1和led2的闪烁，我们可以在两个任务当中都调用vtaskdelay，就可以在led1on后，赶task1阻塞，切换到任务2执行让led2on，此时也就实现了led1和led2的“同步工作”——“并行处理”

（4）**多任务**——对于多优先级的时候，我们对于更高优先级的任务也使用阻塞延时的方法，让低优先级的任务也能“分一杯羹”，在高优先级任务阻塞期间去执行低优先级的任务

（5）因此任务的函数必须是一个死循环，同时死循环当中要有延时阻塞，以此实现任务调度




## 创建任务的总结

### 1、静态创建任务

（1）准备工作

- 用户指定任务栈
- 创建TCB
- 创建句柄
- 创建任务入口（任务的执行函数）
- 起名
- 指定优先级

（2）函数接口

```
TaskHandle_t xTaskCreateStatic( TaskFunction_t pxTaskCode,
                                const char * const pcName,
                                const uint32_t ulStackDepth,
                                void * const pvParameters,
                                UBaseType_t uxPriority,
                                StackType_t * const puxStackBuffer,
                                StaticTask_t * const pxTaskBuffer )
```

返回句柄用于管理任务

（3）函数主要完成的工作

Ⅰ将用户指定的栈完成初始化

Ⅱ任务/TCB初始化（链表、任务名、优先级…）

Ⅲ将任务挂载到就绪列表（初次创建任务会完成就绪列表的初始化，同时会判断新挂载任务优先级更新当前优先级，并完成任务切换）

```
prvInitialiseNewTask( pxTaskCode, pcName, ( uint32_t ) usStackDepth, pvParameters, uxPriority, pxCreatedTask, pxNewTCB, NULL );
prvAddNewTaskToReadyList( pxNewTCB );
```

Ⅱ、Ⅲ接口：

<其中TCB初始化完成的任务之一就像前面所述，将TCB和任务栈连接>

### 2、动态创建任务

（1）函数接口

```
/* 创建AppTaskCreate任务 */
xReturn = xTaskCreate((TaskFunction_t )AppTaskCreate,  /* 任务入口函数 */
                      (const char*    )"AppTaskCreate",/* 任务名字 */
                      (uint16_t       )512,  /* 任务栈大小 */
                      (void*          )NULL,/* 任务入口函数参数 */
                      (UBaseType_t    )1, /* 任务的优先级 */
                      (TaskHandle_t*  )&AppTaskCreate_Handle);/* 任务控制块指针 */
```

**补充点：栈的生长方式个TCB与栈的创建次序问题**

（2）函数主要完成的任务

主要的差异：

1、根据堆栈的生长方式给TCB和任务栈动态分配内存

2、随后依旧是完成任务/TCB初始化、将任务挂载到就绪链表（上述）

## 内核启动方法

1、万事俱备只欠东风法

2、过河拆桥法