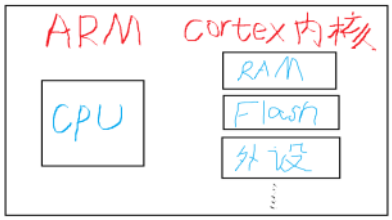
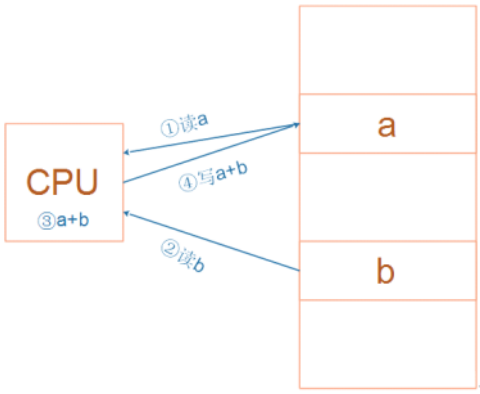
1.任务管理

2025年9月3日 10:52

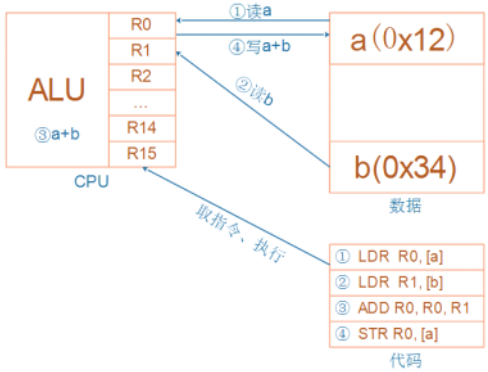
- 一.任务前言
  - 1.1 什么是任务
    - 在freertos中，任务是系统调度和执行的基本单位，本质是一段独立的，可被调度器管理的程序代码
    - freertos中的任务不仅仅是函数，任务是由一个个函数与系统功能的组合
    - 要深入理解任务的本质，就要了解其底层代码的实现，即ARM架构和汇编代码
  - 1.2 ARM架构本质
    - ARM架构不是具体的处理器芯片，而是一套处理器设计规范和指令集体系
    - 它定义了处理器的指令集，寄存器结构，内存访问方式，异常处理机制等底层规则



    - 对于内存RAM只有读和写两个指令，里面存放着各种数据，掉电后数据丢失
    - Flash闪存里面存放着代码(数据的运算)，掉电不丢失
    - 以计算a = a + b为例，看一下如何实现的



      - 1.明显，cpu先从内存当中读取需要数据的值，把值存进cpu的寄存器当中
      - 2.在cpu中进行a+b的运算
      - 3.把计算后的值写回a中
    - 图中过程较为简略，并没有说a和b是怎么被读的，读了之后放在cpu哪里，怎么进行计算的，以及怎么将值返回到a中的。接下来我们将要深入ARM处理器内部的代码实现：



      - cpu中有许多寄存器，将a的值读到寄存器R0中，然后b读到R1寄存器中，然后cpu从Flash中读取指令（汇编指令）（右下角是Flash中的汇编指令），然后执行a+b的操作，计算完成后将a+b的值写入到RAM中对应a的位置，关于Flash中的汇编指令，这里就不多介绍了。
- 二.内存管理机制

- 2.1 前言
  - 我们平时所说的堆栈其实是两个不同的概念，它们分别有不同的作用和含义。
- 2.2 内存的种类及含义
  - 2.2.1 堆(Heap)，是一块动态分配的内存区域，在程序运行是申请和释放空间
  - 2.2.2 栈（Stack），是一种先进后出（LIFO）的内存区域，用于临时存储函数调用和任务运行时的局部数据和任务环境。
- 2.3堆的作用
  - 2.3.1动态分配内存
    - 1.在创建任务时动态分配任务控制块（TCB）
    - 2.动态创建队列，信号量，事件组等内核对象
    - 3.程序中需要管理的大数据块（如缓冲区）
  - 2.3.2FreeRTOS中堆的实现
    - FreeRTOS提供了5种堆的实现方式分别是Heap_1.c到Heap_5.c这几种方案，我们平时用到的是第四种方案Heap_4.c，这5种方案分别对应不同的效果，可以按需选择。
- 2.4堆的特点
  - 堆的分配和释放时动态的，通过Config.h中的configTOTAL_HEAP_SIZE 来配置大小
  - 需要调用API函数分配，其中分配（pvPortMalloc()）和释放（vPortFree()）
- 2.5栈的作用
  - 2.5.1 任务栈
    - xTaskCreate()创建任务时完成，栈空间从FreeRTOS管理的动态内存堆（Heap）中分配。
    - 每个FreeRTOS任务都需要独立分配栈空间（在创建任务时xTaskCreate()时，参数usStackDepth）
    - 存储任务函数中的局部变量
    - 存储函数调用的返回地址
    - 函数的参数和寄存器上下文（任务切换时，FreeRTOS会自动将当前任务寄存器状态保存到对应的栈中）
  - 2.5.2 中断栈
    - 只有部分单片机才有，用与中断服务程序（ISR）的临时数据储存，避免占用任务栈的空间。
- 2.6栈的特点
  - 栈的大小是固定的（任务创建时指定），超出会导致栈溢出，可能引发程序崩溃。
  - 有编译器自动管理（分配和释放），无需手动操作。
- 2.7 堆（Heap）和栈（Stack）的总结
  - **总结：堆 vs 栈**

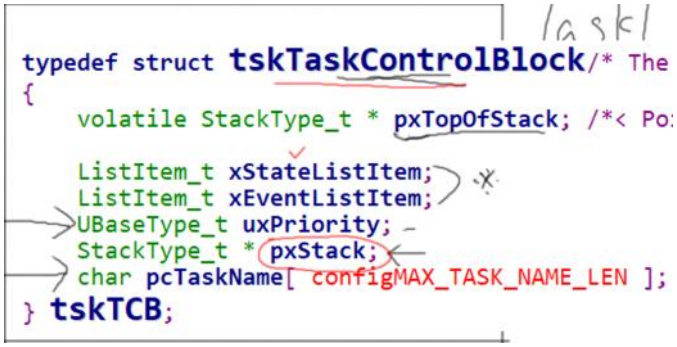
特性	栈（Stack）	堆（Heap）
分配方式	编译器自动分配 / 释放	手动调用函数（pvPortMalloc 等）
大小	固定（任务创建时指定）	动态（总大小编译时配置）
用途	任务运行时的局部变量、函数调用上下文	动态创建内核对象、大数据块等
速度	快（直接操作内存地址）	较慢（需要查找可用内存块）
风险	栈溢出（需预留足够空间）	内存泄漏、碎片化（需正确释放）

三.任务的创建

- 2.1 任务创建的两大核心
  - 2.1.1 任务栈
    - 任务栈用于存储任务运行过程中的临时数据，比如函数调用时的局部变量，函数参数，返回地址，以及任务上下文等信息，它就像时任务运行时的“临时仓库”。
    - 1.存储任务上下文：当任务被暂停（列如发生任务切换）时，当前任务的相关寄存器的值（如程序寄存器pc，状态寄存器，通用寄存器等）会被保存到任务栈中，这就是任务栈的上下文。当任务恢复运行时，再次从任务栈中恢复这些寄存器的值，使任务能够接着暂停状态前的状态继续运行。
    - 2.支持函数调用：在任务执行过程中，每次函数的调用时，函数的参数，局部变量等都需要在栈上分配空间。
  - 2.1.2 任务结构体（TCB）
    - 2.1.2.1 TCB的核心作用
      - FreeRTOS内核无需直接操作代码，只需通过读取/修改TCB中的信息即可。
      - 1.确定任务的优先级，实现“高优先级任务先执行”的调度逻辑。

- 2.保存任务的运行状态（就绪，阻塞，挂起），决定任务是否参与调度。
- 3.记录任务栈的地址和指针，确保任务切换时能正确恢复上下文（寄存器值，程序计数器）。

2.1.2.2 TCB包含的关键信息（核心）



- 1.任务优先级（uxPriority）：值大优先级高，值小优先级低。
- 2.任务栈指针（pxTopOfStack）：指向任务栈的“栈顶”地址，任务切换时，内核会将当前任务的寄存器值压入该栈，恢复任务时再从对应栈中弹出数据。
- 3.任务名称（pcTaskName）：主要用于调试
- 4.链表节点（xStateListItem/xEventListItem）:使TCB能接入FreeRTOS的链表（“就绪链表”，“阻塞链表”），调度器通过操作链表快速管理任务。
- 5.记录任务栈信息：其中pxTopOfStack指针指向任务栈顶位置，pxStack指针指向任务栈起始位置。通过这两个指针，FreeRTOS能够精准定位任务栈的范围，在任务切换等操作时对栈进行管理。

既然栈是保存任务寄存器的地方，那我们该怎么找到这些栈呢，这就是任务结构体的工作。

2.2 动态任务创建的本质

- 一个任务对应一个TCB任务结构体，FreeRTOS内核通过维护所有任务的TCB列表，实现对多任务的统一调度与管理，TCB任务结构体是任务与内核交互的唯一“接口”。
- 任务创建时对应创建一个TCB任务结构体，分配一个任务栈，任务栈中保存着函数的地址，参数，以及寄存器的值。

2.3 任务的创建（动态创建）

- API函数xTaskCreate()

- 第一个参数是TaskFunction_t（任务入口函数的类型定义（函数指针类型））类型的参数 pxTaskCode指向一个指向任务的实现函数的指针（效果上仅仅是函数名）
- 第二个参数是const char*类型的pcName：具有描述性的任务名，这个参数不会被FreeRTOS使用，只是单纯的用于辅助调试
- 第三个参数是const configSTACK_DEPYH_TYPE 定义的栈大小usStackDepth：任务栈根据实际的任务需求配置（局部变量，函数调用深度），过小会导致栈溢出，过大会浪费内存。主要此参数的单位是字（Word）（1字 = 4字节）。
- 第四个参数是void *类型的pvParameters：传递给任务函数的参数指针，传入NULL表示不需要传递参数。
- 第五个参数是UBaseType_t类型的任务优先级uxPriority：指定任务执行的优先级。优先级的范围从最低优先级0到最高优先级（configMAX_PRIORITIES-1）。值大优先级高，值小优先级低。
- 第六个参数是TaskHandle_t类型（标识和操作任务的句柄类型）的pxCreatedTask：此参数是向外输出的参数，用于返回创建的任务句柄，操作任务函数。如果无需操作函数则传入NULL。
- 函数返回值：创建完成之后函数会返回一个值用于表示任务是否创建成功。返回pdPASS表示任务创建成功；返回errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY表示内存不足(无法为任务栈和任务控制块TCB分配空间)

2.4 TCB结构体，任务栈，任务环境以及相关寄存器之间的联系

- 在任务创建时，FreeRTOS会为任务分配TCB结构体和任务栈空间。TCB结构体中的指针指向任务栈，用于管理任务栈的使用。同时，当任务第一次运行或者发生任务切换时，当前任务的相关寄存器的值会被保存到任务

栈中，构建起任务的上下文，这些寄存器值连同任务栈的状态等信息，共同构成了任务环境。当需要恢复执行任务时，FreeRTOS会从任务栈中取出之前保存的寄存器值，恢复到CPU的寄存器中，让任务能继续运行。

- 四.任务的调度机制
  - 关于任务调度，我们将从以下几个方面讲解：
  - 4.1 谁来调度
    - FreeRTOS中的任务调度由内核中的任务调度器（Scheduler）来执行，在任务开始执行前就要开启任务调度器，调用函数vTaskStartScheduler()函数开启任务调度。
  - 4.2 调度什么
    - 开启任务调度器之后，任务调度器会对每个任务（包括任务栈空间，程序计数器pc，寄存器状态等上下文呢信息）根据不同的任务调度机制进行调度
    - 调度器的核心工作是对任务的状态切换和CPU使用权分配
      - 4.2.1任务状态
        - 根据上文我们可以知道，任务结构体TCB中包含了两个链表（xStateListItem/xEventListItem），那么这两个链表是用来干什么的呢，接下来我们要深入看一下这两个链表

```
PRIVILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ];
PRIVILEGED_DATA static List_t xDelayedTaskList1;
PRIVILEGED_DATA static List_t xDelayedTaskList2;
PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList;
PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList;
PRIVILEGED_DATA static List_t xPendingReadyList;
```

          - 这是下StateListItem链表中的列表，我们来详细看一下各个列表分别用于什么
          - 1.pxReadyTaskLists[configMAX_PRIORITIES]：从这个列表的名称我们不难理解。这是就绪任务列表，这是一个数组，大小由configMAX_PRIORITIES（最大任务优先级）决定，所以这个列表由许多个不同优先级的列表共同构成，根据任务优先级的不同，把不同优先级的任务放到不同的就绪任务列表中。
          - 2.xDelayTaskList：很明显这两个的列表是用来管理处于延时状态的任务。FreeRTOS采用“滴答”计数来实现任务延时，任务会根据延时时间被放置到相应到相应的延时列表中。两个延时列表交替使用，用于处理任务的延时到期情况，确保延时任务能在合适的时机重新进入就绪状态。
          - 3.pxOverflowDelayedTaskList：指向另一个另一个延时任务列表，在处理任务延时溢出等特殊情况时使用，保证延时任务管理的完整性。
          - 4.xPendingReadyList：用于暂存那些即将进入就绪任务列表的任务。当任务延时到期，等待的事件发生等情况时，任务会先被放到这个等待处理就绪列表中，之后再被转移到对应的就绪任务列表（pxReadyTaskLists中相应优先级的列表），确保任务状态切换的有序性。
          - 关于任务事件链表，因为与任务创建的关系不大，所以这里就不过多讲述。到后面讲述事件的时候再详细说明。
        - 所以任务调度的核心是通过管理任务列表实现任务的状态切换，将任务从一个列表放到另一个列表中。操作的本质是根据任务状态变化，将任务节点在这些任务之间迁移。
        - 注意：**FreeRTOS总是从就绪状态任务列表中寻找任务执行，当一个任务从就绪状态列表被移到阻塞或者从阻塞被移到暂存任务列表时，此任务都不会再被系统寻找与执行，只有当任务再次回到原来的就绪态列表时才会被寻找与执行。**
      - 4.2.2 CPU的分配：
        - 从就绪任务列表中选择“最合适”的任务，将CPU的使用权分配给它，并完成任务上下文的切换。
  - 4.3 怎么调度（任务调度机制）
    - FreeRTOS中的任务调度机制有许多种，采用基于优先级的抢占式调度，辅以时间片轮转，下面将对各种任务调度机制进行详细说明：
      - 1.核心调度策略：
        - 优先级抢占：调度器始终选择当前就绪态任务列表中优先级最高的任务执行。若最高优先级的任务从就阻塞态转为就绪态的时候，调度器会立即暂停当前运行的低优先级任务（低优先级任务会把此时的任务状态保存到任务栈中，即任务上下文），切换到高优先级的任务。
        - 时间片轮转：对于相同优先级的任务，调度器采用时间片轮转（需开启configUSE_TIME_SLICING）。每个任务执行一个时间片（一个系统时钟节拍Tick周期）后，调度器切换到下一个同优先级的任务。
        - 空闲任务礼让：如果同时优先级为0的其他任务处于就绪状态，空闲任务会主动放弃一次运行机会。
      - 2.调度触发的时机：
        - 调度器并非时刻运行，而是在特定事件发生时触发
          - 任务主动放弃CPU使用权以及被动放弃：当任务调用阻塞型API函数（如vTaskDelay（），队列的

接收函数xQueueReceive（）等）进入阻塞态，就会触发调度器根据任务调度机制进行任务调度。

- ◇ 当优先级相同时，一个时间片运行结束，由系统触发Tick中断进行任务轮转
- ◇ 任务状态主动改变：如调用taskYIELD（）主动让出CPU资源，或者调用vTaskPrioritySet（）修改优先级任务，会导致任务触发调度器进行任务调度。
- ◇ 外部事件触发：任务因等待事件而进入阻塞时（如队列的读取或者写入），当事件发生或者条件满足时，触发任务调度器，任务从阻塞态列表被调度器移动到就绪态列表。

□ 3.调度的具体操作

- ◆ 当调度器被触发时，会执行以下关键步骤：
  - ◇ 1.从就绪任务列表中选取任务优先级最高的任务（若有多个优先级相同的任务，则按时间片或就绪顺序选择）。
  - ◇ 2.若选中的任务与当前运行任务不同，则执行上下文切换：
    - ▸ 保存当前任务的上下文（寄存器，程序计数器等）到其任务栈
    - ▸ 从新任务的栈空间恢复其上下文，更行CPU寄存器
    - ▸ 跳转到新任务的执行地址，开始执行

□ 补充：系统时钟中断（Tick）

- ◆ 对于Tick中断认识
  - ◇ 首先Tick中断是由硬件定时器（SysTick定时器）周期性触发的中断，时FreeRTOS的“心跳”
  - ◇ 两次Tick中断的之间的时间间隔被称为Tick周期，通常在FreeRTOSConfig.h中的configTICK_RATE_HZ设置

在 FreeRTOSConfig.h 中配置：

◇

```c
// 配置Tick频率，例如1000Hz表示1ms一个tick
#define configTICK_RATE_HZ                      1000UL
```

  - ◇ 一个Tick周期 = 1 / configTICK_RATE_HZ秒
- ◆ Tick中断的主要作用
  - ◇ 在每个Tick中断时都会检查任务的状态，决定是否进行任务切换。对于相同优先级的任务，默认情况下开启时间片轮转（configUSE_TIME_SLICING为1），即在每个Tick中断之后都会进行任务切换。不同优先级的任务在会进行判断任务是否需要进行切换。**但是并不一定每个任务都是在Tick中断的时候进行任务切换，当任务主动让出CPU资源时，无需等到Tick中断触发即可进行任务切换。进行任务切换时首先会取出下一个Task（根据任务优先级，从就绪态任务优先级最高的任务列表开始依次向下寻找），然后保存当前Task，恢复下一个Task。**
  - ◇ Tick中断与延时函数的计数有着直接关系，当任务调用vTaskDelay()时，会在指定的tick数后被重新唤醒
  - ◇ Tick中断同时为队列，信号量等同步机制提供超时判断依据
  - ◇ Tick时钟为系统提供了运行时间参考。

□ 4.通过创建任务深入理解调度机制

- ◆ **假如我已经创建了三个优先级都为0的任务**
- ◆

```
xTaskCreate( (TaskFunction_t) Task1_Start,
             (char *) "Task1_Start",
             (configSTACK_DEPTH_TYPE) START_TASK1_DEPTH,
             (void *) NULL,
             (UBaseType_t) START_TASK1_PRIORITY,0
             (TaskHandle_t *) &Start_Task1_Handle);
xTaskCreate( (TaskFunction_t) Task2_Start,
             (char *) "Task2_Start",
             (configSTACK_DEPTH_TYPE) START_TASK2_DEPTH,
             (void *) NULL,
             (UBaseType_t) START_TASK2_PRIORITY,0
             (TaskHandle_t *) &Start_Task2_Handle);
xTaskCreate( (TaskFunction_t) Task3_Start,
             (char *) "Task3_Start",
             (configSTACK_DEPTH_TYPE) START_TASK3_DEPTH,
             (void *) NULL,
             (UBaseType_t) START_TASK3_PRIORITY,0
             (TaskHandle_t *) &Start_Task3_Handle);
```

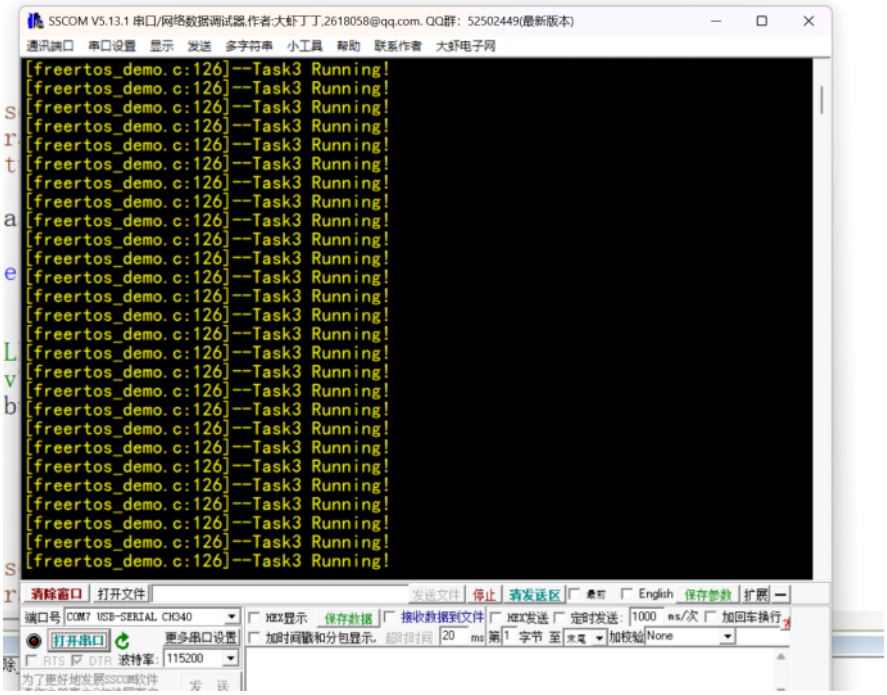
- ◆ 那么再任务全都创建完成之后谁会是第一个执行的呢？我们运行代码看一下
- ◆

```
[freertos_demo.c:83]--Task1 Running!
[freertos_demo.c:100]--Task2 Running!
[freertos_demo.c:126]--Task3 Running!
```

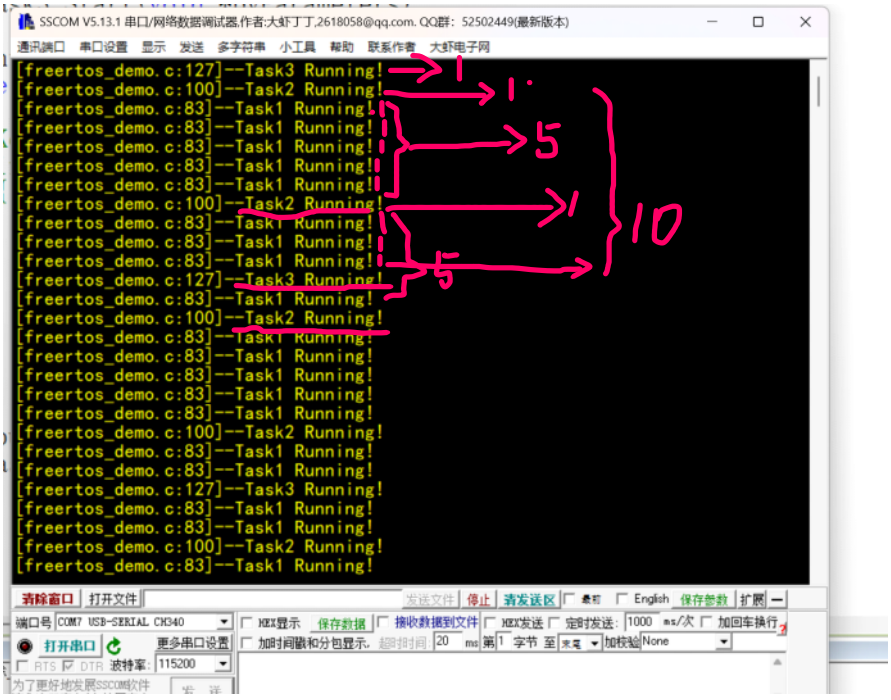
- ◆ 通过运行结果我们可以看到先运行的任务是Task1，然后是Task2，最后是Task3
- ◆ 所以我们可以知道，在FreeRTOS中如果创建任务的优先级相同和默认配置的情况下，先创建的任务会

先执行，之后这三个任务会按照时间片轮转的顺序执行。

- **如果三个任务优先级不同的情况下，Task1优先级为2，Task2优先级为3，Task3优先级为4，（不加延时和阻塞的情况下）我们观看一下运行的结果：**



- 很明显可以看出来，**在不加阻塞和延时的情况下，高优先级的任务Task3一直执行，一直占据CPU的资源，低优先级的任务Task2和Task1根本没有办法得到执行，这就是抢占式调度机制**，那么Task2想要执行就得让Task3让出CPU资源，那么就得让Task3进行延时或者Task3任务进入阻塞状态，同理Task1想要执行，就得让Task2和Task3都让出CPU，让他们都进入阻塞状态！
- **当我为Task3加了10ms的延时，给Task加了5ms的延时，我们再来看看效果：**



- 当任务调度器启动后，因为Task3任务优先级最高，所以Task3最先执行，可是Task3执行到**vTaskDleay（10）；**这句代码时进入**阻塞等待10ms（10个时间片）**（Task3从最好优先级就绪态任务列表被移到阻塞态任务列表，不会被调度器扫描），**此时调度器会从就绪列表中寻找此时列表中优先级最高的任务开始执行**，此时正好Task2的优先级是最高的，所以紧接着Task2开始执行，当Task2也执行到**vTaskDleay（5）；**时，Task2也进入阻塞等待5ms（5个时间片）（Task2也从优先级为3的就绪态列表转移到阻塞态任务列表，不会被调度器扫描），此时调度器又会去就绪任务列表中寻找此时任务优先级最高的任务即Task1，所以Task1执行，当Task1执行完一个时间片后，此时Task2才阻塞了一个时间片，还剩4个时间片，Task3阻塞了里两个时间片，还剩8个时间片。所以此时就绪任务列表中就Task1一个任务，所以Task1一直执行，当Task1在执行4个时间片后，**此时Task2阻塞时间到**（Task2从阻塞态任务列表又回到原来的就绪态任务列表中），此时任务调度器在对就绪态任务列表

进行扫描时，Task2就成为了优先级最高的任务，被执行，然后又进入**vTaskDleay（5）；**进行5个时间片的阻塞延时（再次被转移到阻塞态列表中），然后此时又只剩下Task1一个就绪态任务，Task1又执行3个时间片之后，**此时正好执行了10个时间片的时间**，**此时Task3阻塞时间到**（Task3从阻塞任务列表回到了原来的就绪态任务列表），此时调度器在选择执行任务时就会到Task3执行，然后再次进入10个时间片的阻塞延时。。。。。。然后就是Task2阻塞时间到。。。。。。

- 描述了一遍任务在优先级不同的情况下，任务调度的机制，**就是当就绪态任务列表中有优先级最高的任务时，就执行高优先级任务，阻塞就会让出CPU，到下面优先级的任务执行！**
- 又深入讲述了一遍相信已经对FreeRTOS的任务调度机制又了解，接下来我们来看一下任务调度器的开启。

□ **任务调度器**

- ```
  vTaskStartScheduler();
  ```
- 我们在任务创建完成之后，需要手动开启任务调度器，否则就只会对当个任务执行，不会发生任务切换（效果上类似于裸机程序）任务的入口函数不会被调用，同时与任务相关的API函数部分失效，也不会创建空闲任务，Tick中断是由调度器启动时初始化的，如果不开启调度器，那Tick中断就不会被初始化，无法使用！
- 所以**开启任务调度器时FreeRTOS从“静态库”变成“操作系统”的关键调用**！
- 上面说开启任务调度器就会创建一个空闲任务
- ```
  xReturn = prvCreateIdleTasks();
  ```
- 我们在其底层代码中发现了这么一句代码，是一个空闲任务创建的代码，我们点开看一下

□
```
xIdleTaskHandles[ xCoreID ] = xTaskCreateStatic( pxIdleTaskFunction,
                                                 cIdleName,
                                                 uxIdleTaskStackSize,
                                                 ( void * ) NULL,
                                                 portPRIVILEGE_BIT, /* In effect
                                                 pxIdleTaskStackBuffer,
                                                 pxIdleTaskTCBBuffer );
```

- 确实，开启任务调度器会创建一个空闲任务。

□ **空闲任务**有什么用呢

- 在FreeRTOS中，空闲任务（Idle Task）是系统自动创建的一个特殊任务，**当没有其他就绪态任务需要执行时，空闲任务会占用CPU的时间。**它时FreeRTOS中**多任务机制正常运行的基础保障**，以下是关于空闲任务的详细解释：
- **一.空闲任务的核心作用**
  - ◇ 1.填充CPU空闲时间
    - 当系统中所有任务都处于阻塞态，挂起态或者其他非就绪状态时，空闲任务会自动运行，避免CPU处于“无任务可执行”的状态。
  - ◇ 2.任务资源清理
    - 当任务被删除之后，其**占用的资源无法及时得到释放**（如任务控制块TCB，任务栈空间等），而空闲任务在运行时负责回收这些资源（需配置configUSE_IDLE_HOOK = 1）
  - ◇ 3.**低功耗支持**
    - 在空闲任务中可以实现低功耗逻辑（通过空闲钩子函数），当**系统处于空闲任务时让CPU进入休眠模式**，降低功耗
- 二.空闲任务的**特性**
  - ◇ 1.优先级最低
    - 空闲任务的优先级最低是0（FreeRTOS中的最低优先级），这**确保了FreeRTOS中任何一个优先级>=1的就绪态任务都可以抢占空闲任务的执行权**
  - ◇ 2.自动创建
    - 无需手动创建，在开启任务调度器的时候即调用vTaskStartScheduler（）函数时系统自动创建空闲任务。
  - ◇ 3.栈配置
    - 空闲任务的**栈大小由FreeRTOSConfig.h中的configMINIMAL_STACK_SIZE宏定义**，默认值通常足够简单场景使用，复杂场景可能需要调整。

○ **本次学习是我们开始学习FreeRTOS的必修课程，本次我们从对任务的初步了解，到对ARM架构的深入解析，深入学习了任务的栈和堆，了解了任务在内存分配上的机制，接着**

我们从任务的创建到对底层代码的学习，理解了任务创建的本质，最后我们学习了FreeRTOS中最重要的部分，有关于任务调度机制方面的学习，并通过实践案例辅助理解。有了本次的学习，为我们以后学习FreeRTOS奠定了基础！下一章我们将讲解队列的有关知识，并深入底层取深入了解队列，那本节就到此结束了，我们下一节见！