资深工程师潜心之作，内容兼顾原理和动手实践。

引领读者从原理、设计、实现、应用4个层面仔细体验RTOS的精妙之处。

全书图文并茂，配合STM32硬件平台和源代码，深度践行口袋实验室理念。

单片机与嵌入式

RTOS

嵌入式实时操作系统原理与最佳实践

刘旭明 著

机械工业出版社
China Machine Press

电子与嵌入式系统设计丛书

嵌入式实时操作系统原理与最佳实践

刘旭明 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

嵌入式实时操作系统原理与最佳实践 / 刘旭明著 . —北京：机械工业出版社，2014.8
（电子与嵌入式系统设计丛书）

ISBN 978-7-111-47607-8

I. 嵌… II. 刘… III. 实时操作系统 IV. TP316.2

中国版本图书馆 CIP 数据核字（2014）第 182909 号

本书系统地介绍了嵌入式操作系统内核的原理、设计和实现。首先通过大量图表详细介绍了嵌入式操作系统的基本概念和原理。然后通过对内核各个功能的分析、设计和实现来加深读者对相关知识的理解。最后通过实际的应用程序来演示如何使用这些功能。全书从原理、设计、实现和使用的角度来向读者展示嵌入式操作系统。

本书可作为广大从事嵌入式系统工作的工程师以及其他相关技术人员的参考资料，也可作为相关专业本科生的辅助参考书。

嵌入式实时操作系统原理与最佳实践

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：陈佳媛 责任校对：董纪丽
印 刷： 版 次：2014 年 9 月第 1 版第 1 次印刷
开 本：186mm×240mm 1/16 印 张：22
书 号：ISBN 978-7-111-47607-8 定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

前　言

随着计算机和电子、通信技术的发展，嵌入式系统在人们的生活中变得越来越重要，相关技术发展得越来越快。特别是近几年，各类电子设备得到极快的发展。很多嵌入式系统的开发离不开其核心基础软件，即嵌入式操作系统。本书主要介绍嵌入式操作系统内核的概念与设计实现。

在内容编排上，本书首先对嵌入式操作系统内核各个模块的概念进行讲解，对可选的设计方案进行比较分析，然后采用一个具体的方案作为目标来设计实现，通过流程图、图表、示例代码等详细演示如何实现该机制。最后，还会通过实际应用程序演示这些功能的使用方法。本书从原理、设计、实现和应用各个角度完整展示嵌入式操作系统内核的相关功能。

全书主要内容如下：

第 1 章介绍了嵌入式多任务系统的基本知识。图文并茂地阐述了相关的重要概念。这部分是理解后面章节的基础，初学者需要仔细理解。

第 2 章详细介绍了与任务相关的概念、设计和实现。任务是实时操作系统的重要概念，本章做了十分详细的分析。

第 3 章详细介绍了实现 IPC 机制的基础代码。本章是第 4 ～ 8 章的基础。全书介绍了用于任务间、任务和中断处理函数间的最基本的几个同步和通信机制。

第 4 章详细论述了信号量的设计和实现。

第 5 章详细论述了互斥量的设计和实现。

第 6 章详细论述了邮箱的设计和实现。

第 7 章详细论述了消息队列的设计和实现。

第 8 章详细论述了事件标记的设计和实现。

第 9 章详细介绍了定时器的机制和设计实现。

第 10 章分析了内核的移植代码，着重介绍了在意法半导体的 STM32 处理器上如何移植 Trochili RTOS。读者需要对基于 ARM Cortex-M3 的处理器有一定了解。

第 11 章重点介绍了基于 Trochili RTOS 的以太网协议应用。通过实际案例演示 Trochili

RTOS 的使用。

作者从事嵌入式工作多年，参与过多款嵌入式处理器的功能验证和固件开发工作，经常接触 RTOS，对其有浓厚兴趣。工作之余尝试编写一些任务调度代码，逐渐实现了一套较完整的 RTOS 内核。这个兴趣爱好加深了作者对嵌入式系统的理解，使得个人能力得到了提高，对本职工作也有很大帮助。目前 Trochili RTOS 并不完美，现有很多功能还需优化，作者会逐步完善 RTOS 的各项功能。

由于作者时间和水平有限，书中难免存在错误及不妥之处，敬请广大读者批评指正，如有问题，可通过 Trochili RTOS 官方网站 www.trochili.com 或者微博 www.weibo.com/trochili 和作者联系。

本书在编写和出版过程中得到了机械工业出版社华章公司编辑们的热情帮助和大力支持，在此一并表示感谢。

感谢北京航空航天大学何小庆教授对本书的指导，何老师在全书内容选取和章节安排上给了很好的建议，并对全书做了审读检查。感谢我的朋友王文东，他对全书内容进行了详细的阅读并给出了很多改进意见。

本书可作为广大从事嵌入式系统开发工作的工程师以及其他相关技术人员的参考资料，也可作为相关专业本科生的辅助参考书。

刘旭明

2014 年 4 月于北京

目　录

第 1 章
嵌入式操作系统基础

本章主要介绍多任务嵌入式操作系统相关的概念和整体结构，并对全书涉及的重要知识做介绍，为我们在以后各章的嵌入式操作系统的内核分析和学习做好准备。本章的内容并没有强调、区分嵌入式操作系统和通用操作系统的概念。在这两者之间，很多机制是相通的。另外本章假设读者已经对嵌入式系统有初步了解，不再介绍例如发展历史、机制特点这些基本知识。

1.1 嵌入式软件系统结构

目前常见的嵌入式软件结构可以分为轮询系统、前后台系统和多任务系统。这些方案是根据应用的具体需求提出的，各有各自的特点和适用的领域。

1.1.1 轮询系统

这是最简单的一种软件结构，主程序是一段无限循环的代码，在循环中顺序查询各个条件，如果满足就执行相应的操作。这种方案的好处是实现简单，逻辑清晰，便于开发人员掌握。但是每个事件的查询和处理时间是不能确定的。假如前面的操作时间较长，那么后面的操作必然会被延迟。

在图 1-1 中，假如步骤 1 操作需要很久，那么步骤 2 必然得不到及时处理，如果步骤 2 的工作很重要或者很紧急，那么系统的性能和响应能力就很差了。

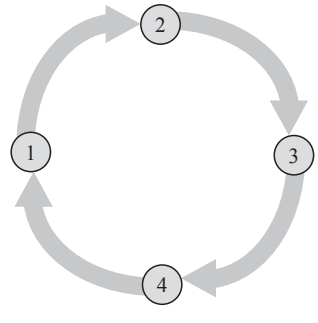

图 1-1 轮询系统结构

1.1.2 前后台系统

相对轮询系统，前后台系统对外部事件的处理做了优化。前后台系统是由中断驱动的。主程序依然是一段无限循环的代码，称为后台程序，而事件的响应则由中断来完成，称为前台程序。在后台程序执行时，如果有外部事件发生，则前台的中断程序会打断后台程序。在完成必要的事件响应之后，前台中断程序退出并通知后台程序来继续操作。由后台程序完成

事件的后继处理，比如数据的分析等操作。从代码功能上讲，事件的响应和处理分为了两个部分。因为中断自身有优先级和嵌套的功能，所以优先级高的事件能够得到及时响应。但后台程序仍然需要按顺序处理各个事件的后继事务。

前后台系统演示如图 1-2 所示：

如图 1-2 所示，在中断源之间有优先级的概念。ISR 会首先响应事件，简单的事件可以在 ISR 中直接处理，复杂的情况下则记录下必要数据和状态标记，等所有中断处理结束后，将由后台主函数按顺序处理各个事件。也就是说，事件的响应是支持优先级的，但事件的最终处理却是顺序的。使用中断来代替轮询方案中事件的查询操作，所以相对轮询方案，前后台系统对事件的响应能力有较大改善。

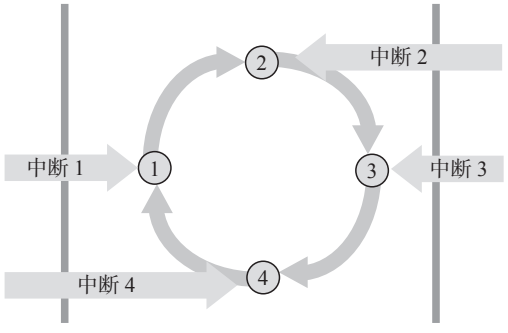

图 1-2 前后台系统结构

1.1.3 多任务系统

和前后台系统相比，多任务系统在响应事件的时候，同样是由多个中断处理程序完成的。但是对于事件的后继操作则是由多个任务来处理的。也就是说每个任务处理它所负责的事件。在基于优先级的多任务系统中，因为任务间优先级的关系，优先级高的任务可得到优先处理。这样优先级高的事件就能及时得到处理；在基于分时机制的多任务系统中，任务间则按比例轮流占用处理器。

多任务机制如图 1-3 所示。

在图 1-3 中，中断用来响应事件，事件的后续操作则由任务来完成。中断和任务都有优先级。假如其中中断 2 和任务 2 处理的事件是紧急的或者重要的，那么当中断 2 发生时，即使其他任务或者中断正在处理，也会被抢占，最终任务 2 会优先得到运行机会。

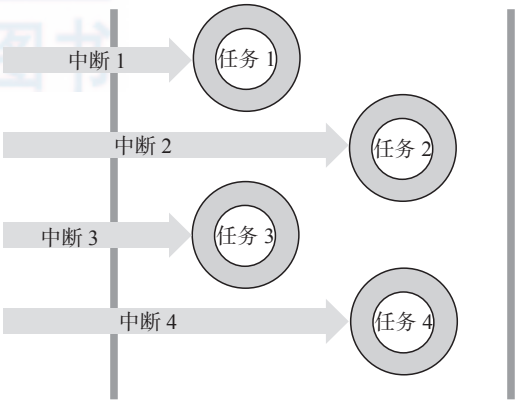

图 1-3 多任务系统结构

因为多任务操作系统允许将具体的应用系统分成若干个相对独立的任务来管理，所以多任务操作系统的使用可以简化应用程序的设计，系统也变得简洁且便于维护和扩展。对实时性要求严格的事件都能得到及时可靠的处理。不过多任务操作系统自身将占用部分处理器、存储器等硬件资源，这是引入多任务机制的必要代价。

从事件和数据处理的角度考虑，可以把整个应用流程简化为事件响应和事件处理两个阶段。从这两个阶段采用的不同技术手段出发，可以清晰合理地分析上面介绍的这三种软件结

构方案，可以看到解决问题的思路越来越清晰，结构和层次越来越合理。

表 1-1 是对三种软件结构的比较。

表 1-1 常见嵌入式软件模型

模型	事件响应	事件处理	特点
轮询系统	主程序	主程序	轮询响应事件，轮询处理事件
前后台系统	前台多个中断程序	后台单个主程序	实时响应事件，轮询处理事件
多任务系统	多个中断程序	多个任务	实时响应事件，实时处理事件

通过上面的比较，我们可以清楚地看到嵌入式软件结构上的不同和发展，但这并不是系统结构好坏的标准。每种方案都有它产生的年代、硬件资源的发展阶段和所适合的应用领域。

多任务系统是基于多任务操作系统的应用开发模型。本书介绍的就是嵌入式操作系统的核心部分：嵌入式操作系统内核的设计和实现。它的主要功能包括：任务管理、任务调度、任务同步、互斥和通信、设备管理、中断管理、时间管理等。而像图形用户接口、文件系统、TCP/IP 协议、嵌入式数据库引擎等，则可以归为嵌入式操作系统内核层之外的功能模块。多任务模型下 RTOS 组成如图 1-4 所示。

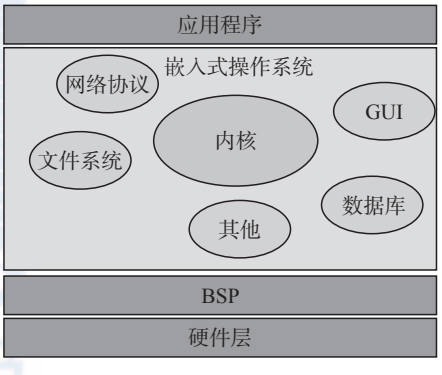

图 1-4 多任务模型下 RTOS 组成

关于嵌入式操作系统，有很多常见的技术概念，熟悉这些概念是我们学习嵌入式操作系统的基础。本书后续章节着重分析、设计和实现一个“嵌入式实时操作系统内核”，有时会使用“内核”这个简称。在内容的编排上，会把各种功能模块的概念放在各章起始，首先介绍其原理，然后分析设计和实现。

1.2 多任务机制概述

在前面我们介绍了多任务系统是如何演化的。和前后台系统相比较，多任务可以理解为有多个后台程序的前后台系统。每个任务都专注于自己处理的问题。下面将详细介绍一下和多任务系统相关的一些基本概念。

1.2.1 时钟节拍

时钟节拍是多任务系统的基础，它指明了把处理器时间以多大的频率分割成固定长度的时间片段。作为多任务系统运行的时间尺度，时钟节拍是通过特定的硬件定时器产生的。硬件定时器会产生周期的中断，在相应的中断处理函数中，内核代码得以运行，从而进行任务调度和定时器时间处理等内核工作。

处理器的时钟节拍如图 1-5 所示。

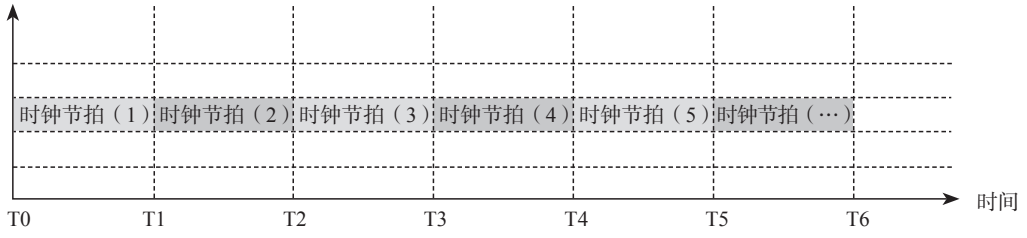

图 1-5　处理器时钟节拍

硬件定时器中断的时间间隔取决于不同的内核设计，一般是毫秒级的。时钟节拍越快，内核函数介入系统运行的几率就越大，时钟节拍中断响应次数越多，内核占用的处理器时间越长。相反，如果时钟节拍太慢，则导致任务的切换间隔时间过长，进而影响到系统对事件的响应效果。

图 1-6 演示了多任务系统中，中断处理程序和任务在时间上的关系。

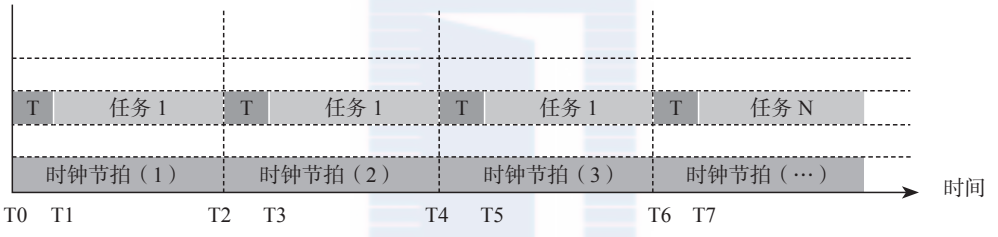

图 1-6　多任务系统中的中断和任务的时间关系

如图 1-6 所示，硬件定时器按照固定的时间间隔产生中断，然后在时钟节拍中断 ISR 中（图中以 T 标记）处理内核的工作。T0 ～ T1 这段时间是内核占用的时间（时钟节拍处理程序），T1 ～ T2 这段时间是任务占用的时间。而 T0 ～ T2 则是一次时钟节拍的全部时间。从图 1-6 中可以看出，任务 1 的本轮执行占用了 3 个时钟节拍。

1.2.2　多任务机制

在单处理器的计算机系统中，在某一具体时刻处理器只能运行一个任务，但是可以通过将处理器运行时间分成小的时间段，多个任务按照一定的原则分享这些时间段的方法，轮流加载执行各个任务，从而从宏观上看，有多个任务在处理器上同时执行，这就是单处理器系统上的多任务机制的原理，如图 1-7 所示。

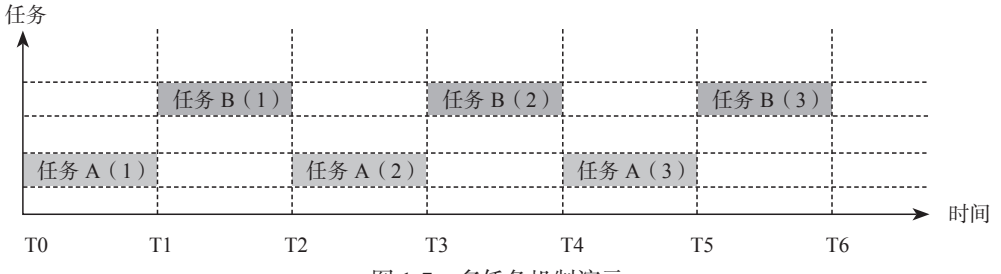

图 1-7　多任务机制演示

在图 1-7 中，任务 A 和任务 B 按照等长时间轮流占用处理器，在单处理器上造成多个任务同时运行的假象。

另外，因为不同任务的运行路径不同，在某一时刻有些任务可能需要等待一些资源，这时可以通过某种方案，使当前任务让出处理器，从而避免因为任务等待资源而长期占有处理器而使其他任务无法运行。这样多任务机制可以使处理器的利用率得到提高，并提高了系统的处理能力。

在多任务操作系统内核中必须提供解决并发任务的机制。通用操作系统一般以“进程”、“线程”等为单位来管理用户任务。在相关资料中，也会明确指出“进程”与“线程”的区别。但在很多嵌入式操作系统中，并没有区分进程和线程，只是把整个操作系统当作一个大的运行实体，其中运行着很多任务。任务通常作为调度的基本单位。

1.2.3 任务上下文

任务可以看作是用户程序在处理器等硬件上的运行，是一个动态的概念。任务在处理器上运行的某一时刻，有它自己的状态，即处理器所有的寄存器的数据，这个叫作任务的上下文，可以理解为是处理器的“寄存器数据快照”。通过这些数据，操作系统可以随时打断任务的运行或者加载新的任务，从而实现不同任务的切换运行。

任务上下文是跟处理器密切相关的概念，不同的处理器有不同的处理器上下文定义。比如在 Cortex-M3 处理器中的寄存器如下所述：

- 拥有 R0 ～ R15 寄存器组。
- R0 ～ R12 是通用寄存器。
- R13 作为堆栈指针（设置有两个，但在同一时刻只有一个指针起作用。）
- R14 为连接寄存器。
- R15 为程序计数器，指向当前的程序地址。

另外还有特殊功能寄存器：

- 程序状态寄存器组（PSR）
- 中断屏蔽寄存器组（PRIMASK、FAULTMASK、BASEPRI）
- 控制寄存器（CONTROL）

在 RTOS 设计任务上下文时经常会把大部分硬件寄存器作为任务上下文的内容，这点在介绍操作系统移植时会做详细介绍。

1.2.4 任务切换

任务切换又叫作任务上下文切换。当操作系统需要运行其他的任务时，操作系统首先会保存和当前任务相关的寄存器的内容到当前任务的栈中，然后从将要被加载的任务的栈中取出之前保存的全部寄存器的内容并加载到相关的寄存器中，从而继续运行被加载的任务，这个过程叫作任务切换。

任务基本的切换过程如图 1-8 至图 1-11 所示。

假设系统中有两个任务 A、B。当前处理器正在运行 A 任务。此时任务 A 的栈顶在变量 An 处：

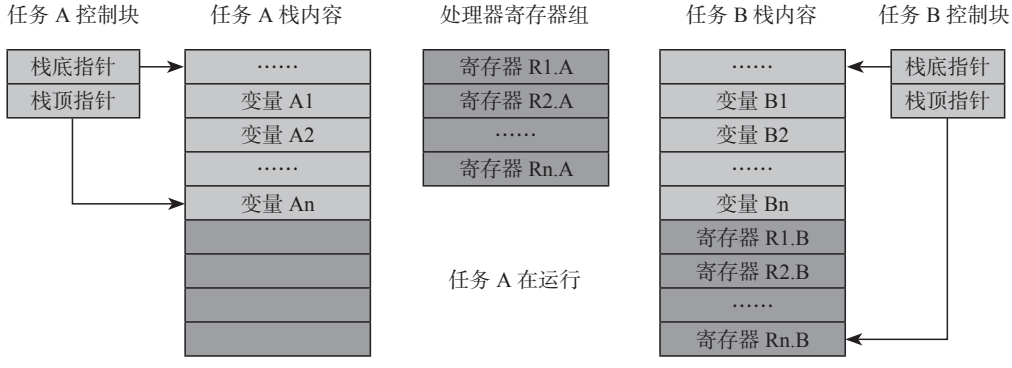

图 1-8　任务 A 运行时的上下文情况

然后发生任务调度，需要首先保存当前的处理器寄存器组的内容到任务 A 的栈中：

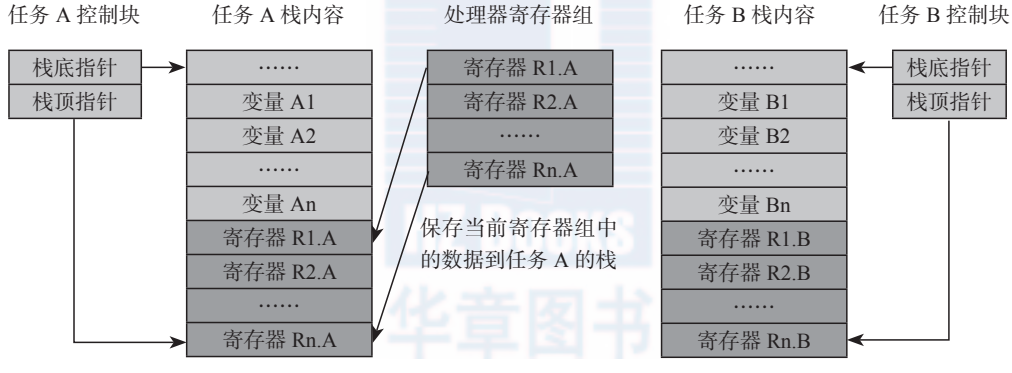

图 1-9　任务 A 上下文保存

接下来的操作就是把保存在任务 B 栈中的处理器寄存器组的内容调入到处理器寄存器组中：

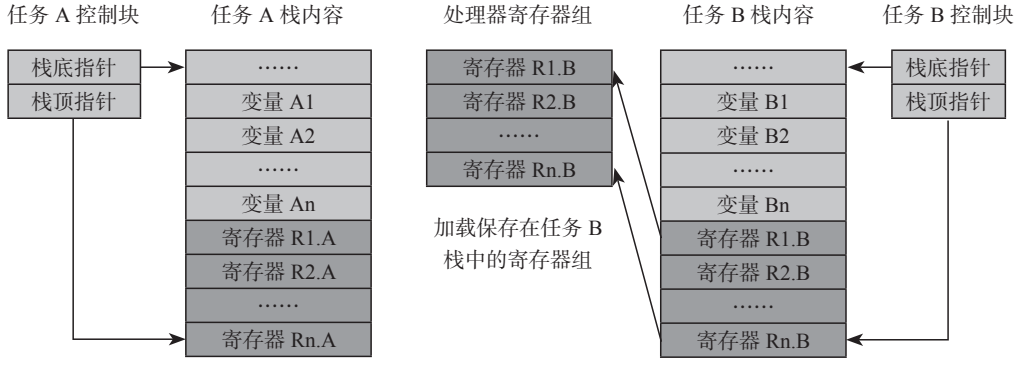

图 1-10　任务 B 上下文恢复

最后，处理器开始继续执行任务 B：

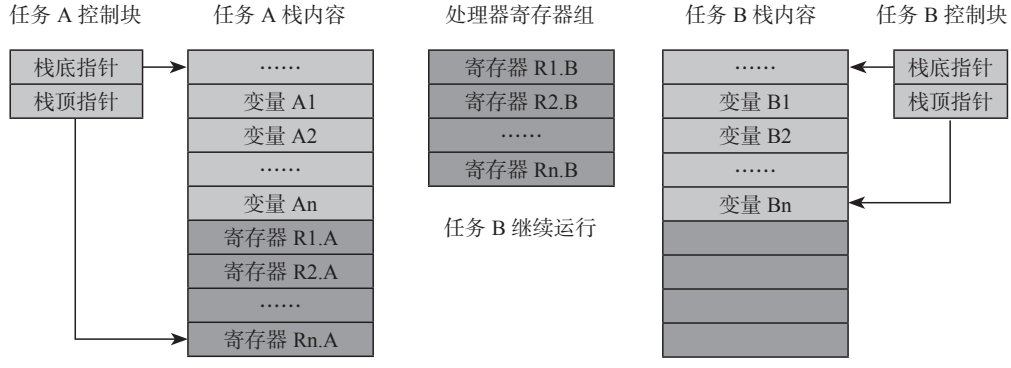

图 1-11　任务 B 运行时的上下文情况

这里给读者留两个问题：

❏ 保存在任务 B 栈中的最初始的寄存器组的内容从哪里获得？

❏ 保存在最先执行的任务栈中的寄存器组的数据是如何加载到处理器寄存器组的？

1.2.5　任务的时间片和优先级

时间片指的是任务一次投入运行，在不被抢占或者中断的情况下，能够连续执行的最长时间（以时钟节拍计数）。时间片的长度由具体操作系统规定，有些操作系统中不同任务可以有不同的时间片长度，或者是在运行过程中可以动态改变时间片长度。

时间片长度是时钟节拍的整数倍，如图 1-12 所示。

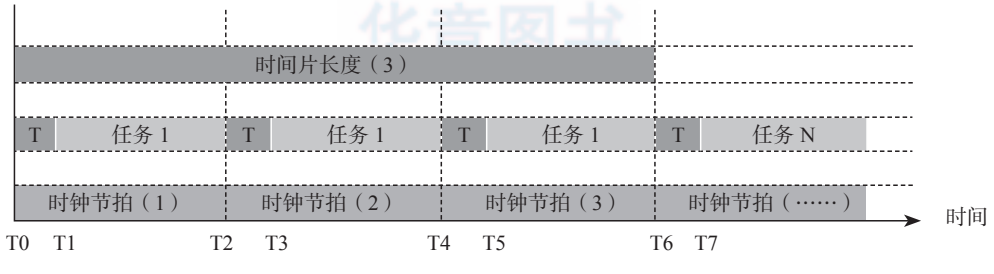

图 1-12　时间片和时钟节拍关系

从上图中可以清楚地看出，任务时间片、时钟节拍、定时器 ISR 之间的时间关系：时间片长度是时钟节拍的整数倍，一个时钟节拍被定时器 ISR 和任务分享。

任务的优先级用于安排系统中各个任务的执行次序，它说明了任务的重要性，任务越重要，它的优先级应越高，越应该获得处理器资源。任务优先级的安排有两种方式：静态优先级和动态优先级。如果任务优先级在运行的过程中不能改变，则称为静态优先级。静态优先级是在任务初始化时决定的；反之如果任务优先级是可以改变的，则称为动态优先级。

时间片和优先级是任务的两个重要参数，分别描述了任务竞争处理器资源的能力和持有

处理器时间长短的能力。这两者同时是任务抢占的重要参数。因任务时间片运行完毕而引起的任务调度可以理解为时间片调度，而因为操作系统中最高就绪优先级的变化而引起的调度则为优先级调度。

1.2.6 任务调度和调度方式

任务调度是操作系统的主要功能之一，在任务需要调度的时候，操作系统会根据具体的调度算法和策略选择合适的任务，替换当前任务占有的处理器等硬件资源。根据调度原理的不同，任务调度方式可分为可抢占式调度和不可抢占式调度两类。

- 对于基于优先级的系统而言，可抢占式调度是指操作系统可以剥夺正在运行任务的处理器使用权并交给拥有更高优先级的就绪任务，让新的任务运行。
- 对于基于分时机制的系统而言，每个任务都能持续占用处理器一段时间，每轮时间用完之后操作系统就剥夺处理器给别的任务来执行。可以把这种调度方式理解为时间引起的抢占。
- 在不可抢占式调度方式下，如果某任务占有了处理器，那它就一直执行，直到它主动将处理器控制权让给别的任务使用。操作系统不会强制它释放处理器资源。

基于优先级的可抢占式调度的实时性好，任何优先级高的任务只要具备了运行的条件，即进入了就绪态，就可以立即得到调度和运行。任务在运行过程中都随时可能被比它优先级高的任务抢占。这种方式的任务调度保证了系统的实时性。

图 1-13 演示了不可抢占式操作系统中的任务执行情况。例子中有三个任务需要运行。系统的执行流程如下：

- 在 T0 时刻任务 C 得到处理器，它开始执行。在它执行的过程中，任务 B 和任务 C 就绪，但是因为操作系统不支持抢占式调度，所以它们只好等待机会。
- T1 时刻任务 C 完成操作，主动让出处理器。操作系统选择任务 B 来运行。
- T2 时刻任务 B 完成操作，主动让出处理器。操作系统选择任务 A 来运行。
- T3 时刻任务 A 完成操作，主动让出处理器。操作系统再次选择任务 C 来运行。

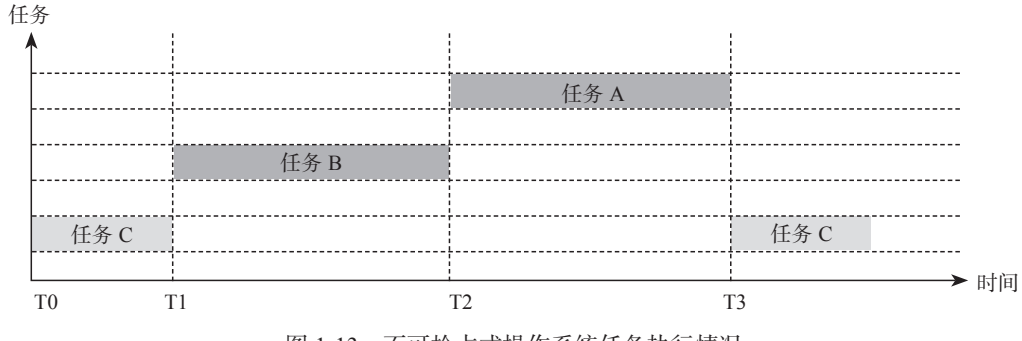

图 1-13 不可抢占式操作系统任务执行情况

对于抢占式调度方式，下面的章节会有详细的图示。

1.2.7 任务调度算法

常见的任务调度机制主要有时间片调度算法（时分式）、优先级调度算法（抢占式）和基于优先级的时间片调度算法。

1. 时间片调度算法

时间片调度算法指的是操作系统先让某个任务运行一个时间片（多个时钟节拍），然后再切换给另一个任务。这种算法保证每个任务都能够轮流占有处理器。因为不是在每个时钟节拍都做任务调度，对处理器资源的消耗又做到了最少。时间片调度算法缺点是在任务占有处理器的时间段内，即使是有更紧急的任务就绪，也不能立刻执行它。

时间片的调度如图 1-14 所示。

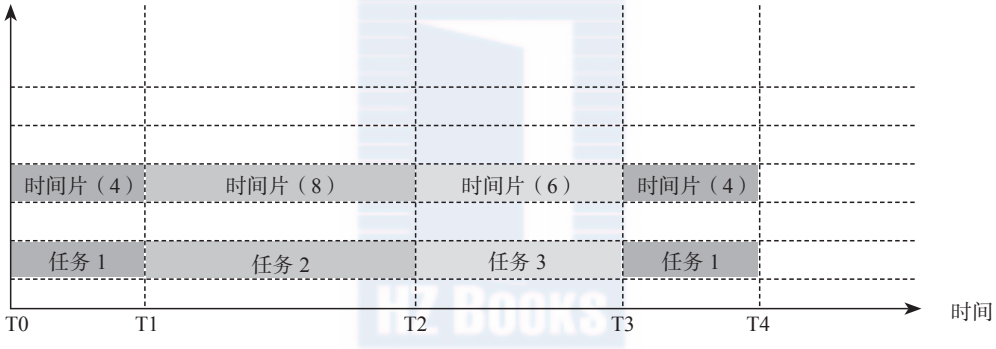

图 1-14 时间片调度算法演示

如图 1-14 所示：

- 假设任务 1 时间片长度为 4；任务 2 时间片长度为 8；任务 3 时间片长度为 6。
- T0 时刻任务 1 得到运行。
- T1 时刻任务 1 时间片结束，本轮运行结束；任务 2 开始运行。
- T2 时刻任务 2 时间片结束，本轮运行结束；任务 3 开始运行。
- T3 时刻任务 3 时间片结束，本轮运行结束；任务 1 开始新一轮运行。

2. 优先级调度算法

优先级调度算法指的是操作系统总是让具有最高优先级的就绪任务优先运行。即当有任务的优先级高于当前任务优先级并且就绪后，就一定会发生任务调度，这种操作系统最大限度地提升了系统的实时性。

优先级调度算法如图 1-15 所示。

图 1-15 只是演示了在时钟节拍中进行优先级调度的情况。

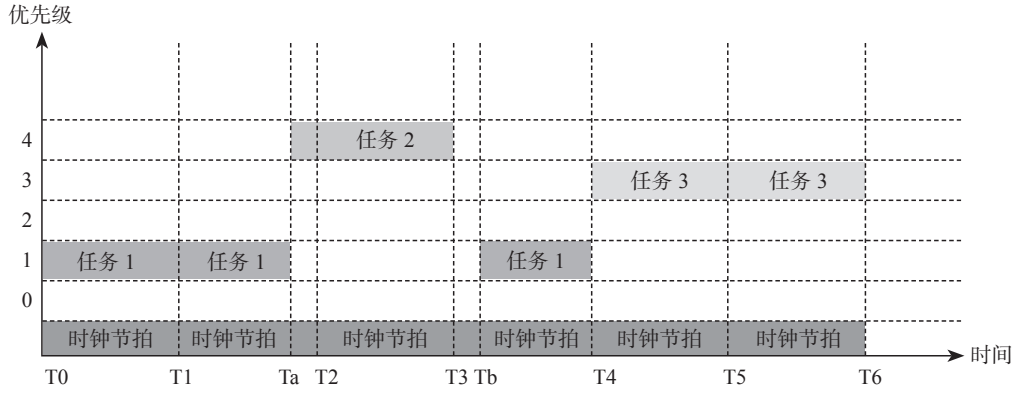

图 1-15 优先级调度算法演示

- 假设任务 1 优先级为 1，任务 2 优先级为 4，任务 3 优先级为 3；数值越大优先级越高。
- T0 时刻系统中只有任务 1 就绪，所以首先得到调度运行。
- T1 时刻因为没有别的任务就绪，所以任务 1 继续运行。
- Ta 时刻任务 2 就绪，因为任务 2 优先级高，所以它抢占任务 1 开始运行。抢占发生在 T1-T2 时间节拍之间。
- T2 时刻因为没有更高优先级任务就绪，所以任务 2 继续运行。
- T3 时刻因为没有更高优先级任务就绪，所以任务 2 继续运行。
- Tb 时刻任务 2 放弃了处理器（比如任务被阻塞），任务 1 得到运行。
- T4 时刻，假设此时在时钟节拍 ISR 里唤醒了任务 3，因为任务 3 优先级高，所以任务 3 抢占任务 1。抢占发生在时钟节拍处。
- T5 时刻，任务 3 继续运行。

优先级调度算法的缺点是，当最高优先级任务在运行时，它将持续占有处理器直到任务结束或者阻塞，否则其他任务无法获得运行的机会。

3. 基于优先级的时间片调度算法

基于优先级的时间片调度算法吸收了以上两种算法的优点，同时又解决了它们的不足。这种算法为每个任务都安排了优先级和时间片。在不同优先级的任务间采用优先级调度算法，在相同优先级的任务间使用时间片调度算法。任务调度策略首先考虑任务的优先级，优先级高的任务必定会抢占低优先级的任务。相同优先级的任务则按照时间片长度比例共享处理器时间。这样既保证了能够尽快响应紧急任务，又保证相同优先级的任务都有机会轮流占有处理器。

该算法如图 1-16 所示。

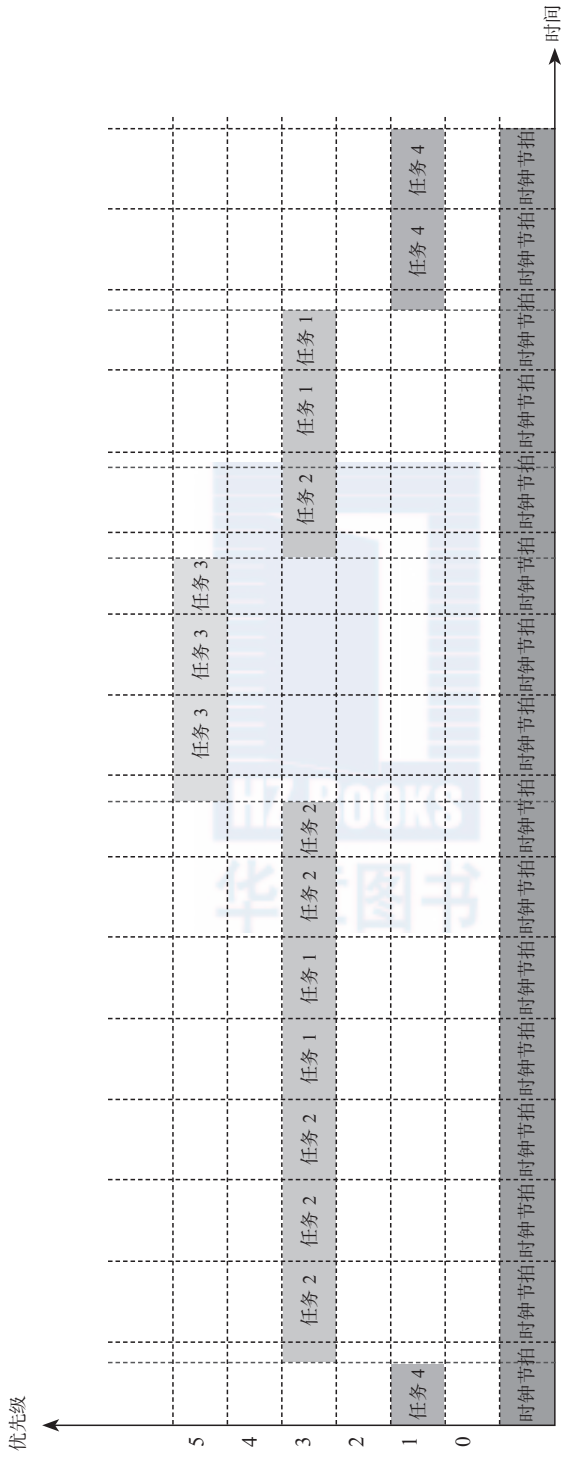

图 1-16 基于优先级的时间片调度算法

如图 1-16 所示：

- 任务 1 优先级为 3，时间片长度为 2 个时钟节拍；任务 2 的优先级为 3，时间片长度为 3；任务 3 优先级为 5，时间片长度为 1 个时钟节拍；任务 4 优先级为 1，时间片长度为 2 个时钟节拍。
- T0 时刻，系统中只有任务 4 就绪，所以任务 4 抢占得到处理器，并运行。
- Ta 时刻，任务 2 和任务 1 就绪，任务 2 因优先级抢占得到处理器，开始运行。
- T2 时刻，任务 2 时间片耗尽，只好释放处理器；任务 1 得到运行的机会。
- T3 时刻，任务 1 时间片耗尽；任务 2 得到运行的机会。
- Tb 时刻，任务 3 就绪，发生优先级抢占；任务 3 得到运行的机会。
- Tc 时刻，任务 3 处理完自己的工作，主动释放处理器；任务 2 得到运行的机会。
- Td 时刻，任务 2 处理完自己的工作，主动释放处理器；任务 1 得到运行的机会。
- Te 时刻，任务 1 处理完自己的工作，主动释放处理器；任务 4 得到运行的机会。

1.2.8 任务状态

在多任务的系统中，任务一般具有多种状态，反映任务不同的执行阶段。常见任务状态主要有以下 3 种：

- 就绪状态：任务已经获得除处理器之外的一切需要的资源，等待任务调度。
- 运行状态：任务正在运行中，获得了所有需要的资源。
- 等待状态：任务缺少某些必需的运行条件或资源而不能参与任务调度。

不同的 RTOS 可能有不同的任务状态定义，可能会更细致地定义一些状态用于管理各个任务。

1.3 同步、互斥和通信

在多任务系统中，在任务间、ISR 和任务间必然存在着处理器交替抢占，轮流执行的情况。除此之外，这些可执行对象也存在着其他关系，仔细观察这些对象，它们总是要“走走停停、互相照应”，这也正是多任务系统的特点，只有这样设计系统才能使得硬件资源得到最大的利用。可以把它们间的关系总结如下。

- 共享资源的竞争：任务或者 ISR 访问共享资源时是互相竞争的，只能被一个任务或者 ISR 访问，并且操作时不能被打断。强调的是“互斥”的概念。
- 运行同步：任务间或者任务和 ISR 间互相协作，按照规定的路线执行，也就是对它们的执行步骤和顺序有要求。强调的是“同步”的概念。同步可以是单向的也可以是双向的。
- 数据通信：任务间或者任务和 ISR 间的数据传输，常见的模式是一方提供数据，另一方处理数据，共同完成某些功能。强调的是“通信”的概念。

任务间的数据传输，可以是直接的，也可以是间接的。

- 在直接数据传输方式下，一个任务可以把数据直接发给指定的任务，发送过程很明确地说明了哪个任务把数据传给了哪个任务。
- 间接方式指的是数据交互的双方，约定一个数据缓冲区，发送数据的任务首先会把数据发往该缓冲区，然后通知接收数据的任务则从该缓冲区取得数据。从操作系统角度来考虑，操作系统不关心这些数据的含意，只当普通的数据来处理。

在本书后面，经常会采用 IPC（Inter-Process Communication）来代表以上各种关系的操作。从字面含义上讲，IPC 这个词并不是很恰当。常见的 IPC 机制包括信号量、邮箱、消息队列、事件集合、条件变量、管道等。

1.3.1 任务等待和唤醒机制

当任务在试图访问 IPC 对象时，经常会因为运行条件不足而失败，被迫返回或者阻塞在该 IPC 对象的任务阻塞队列。而当有任务释放资源从而使得资源条件可以满足时，操作系统将会唤醒 IPC 对象上的阻塞任务，使得被唤醒任务继续运行。不同的访问等待机制和唤醒机制是各种操作系统的重要区别。

用于任务访问 IPC 对象的等待机制主要有三种：

- ❏ 直接返回结果：任务直接返回访问结果，成功或者失败；注意因为 ISR 不像任务那样能够被阻塞，所以 ISR 必须采用本模式。
- ❏ 阻塞等待模式：任务如果访问 IPC 对象失败，则进入该 IPC 对象的等待队列，直到明确得到处理。
- ❏ 时限等待模式：任务如果得不到 IPC 对象，则进入等待状态并开始计时。如果在等待期间得到了 IPC 对象则返回操作成功；如果当计时结束时任务仍然没有成功，那么它并不会继续等下去，而是返回失败的结果。

当任务不能获得资源而进入资源的等待队列之后，如果某个时刻资源可用，那么操作系统就该决定怎么处理这些等待任务。这就涉及操作系统任务唤醒机制。

操作系统唤醒机制主要有以下三种模型：

- ❏ 当资源可使用时，唤醒该资源的全部等待任务。让这些任务与系统中的其他任务平等竞争资源。这种策略会使系统瞬间繁忙，在参与竞争资源的所有任务中，最终只有一个任务获取到资源，没有得到资源的任务将再次进入资源的等待队列。
- ❏ 将该资源等待队列中的一个合适的任务唤醒。这个任务将和系统中可能访问该资源的其他任务一起竞争这个资源。如果这个任务最终没有竞争到资源，它会再次进入该资源的等待队列。
- ❏ 操作系统从等待队列中找到一个最佳的任务并立刻把资源交给它，这样该任务直接从释放资源的那个任务那里获得资源。

目前主流嵌入式操作系统都采用第三种方案。

1.3.2 任务互斥和优先级反转

在系统中，有些资源必须是独占使用的，多个任务对这样的资源的并发访问将导致错误的发生。一般来说，对需要独占使用的资源必须使用互斥方法将对其的并发访问串行化。

在优先级多任务系统中引入互斥方案，会导致任务优先级反转的问题：假如某时低优先级的任务占有资源，然后又有高优先级的任务申请资源，但因为不能满足而被挂起了，即低优先级任务阻塞了高优先级任务的运行。假如这时又有一个中优先级任务，那么它会把低优先级任务抢占。最终高优先级任务会间接地被中优先级任务抢占了。这种现象叫作优先级反转。举例说明：

假如 A、C、D 三个任务优先级从高到低排列，任务 A 和 C 共享互斥信号量 R，如果某一时刻任务 C 已经获得互斥信号量 R，而任务 A 此时尝试占用 R，那么任务 A 会因为得不到 R 而阻塞在 R 的任务等待队列中。再假设此时任务 D 因为优先级高于任务 C 从而抢占了 C，进而长期占有处理器资源，那么就相当于低优先级的任务 D 间接阻塞了高优先级任务 A 的运行。

这个过程可以用图 1-17 来说明。

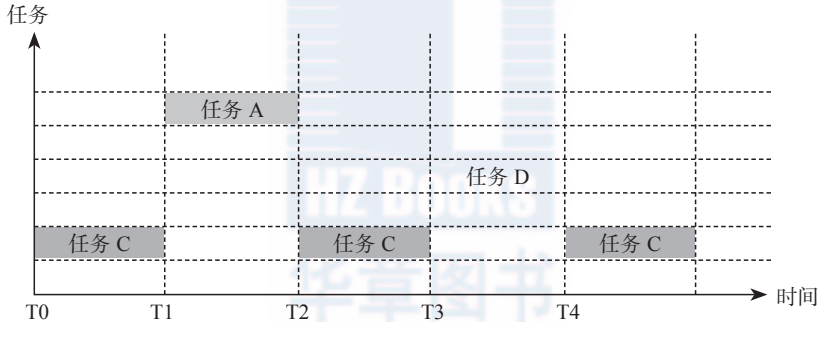

图 1-17 优先级反转

- T0 时刻，任务 C 处于运行状态，运行过程中，任务 C 获得了共享资源 R。
- T1 时刻，任务 A 就绪。由于任务 A 优先级高于任务 C，所以它抢占了任务 C，任务 A 被调度执行。
- T2 时刻，任务 A 需要共享资源 R，但 R 被更低优先级的任务 C 所拥有，所以任务 A 被阻塞等待该资源。任务 C 得到执行。
- T3 时刻，此时任务 D 就绪，由于任务 D 优先级高于任务 C，所以它抢占了任务 C, 任务 D 被调度执行。

从整个流程上看，T3 时刻，高优先级任务 A 被低优先级任务 D 间接地抢占了。此时优先级最高的任务 A 不仅要等任务 C 运行完，还要等优先级低的任务 D 运行完才能被调度，如果任务 D 和任务 C 需要执行很长时间，那么任务 A 的执行就不能得到保证，整个系统的实时性能很差。

优先级反转现象[⊖]对基于优先级调度的实时系统有很大的影响。在基于优先级调度的系统中，处理器资源是按照优先级分配给任务的，就绪的高优先级任务必须实时获得处理器。系统中的各种资源，如果采用按照任务优先级分配的原则，那么高优先级的任务应该是首先被考虑的。优先级反转的问题将打乱这些原则。

1.3.3 优先级天花板和优先级继承

优先级反转问题的核心原因在于共享资源的访问规则，即共享资源只能被一个任务占用，被占用后其他任务不能强制使用这个资源。在优先级反转问题上，高优先级任务被低优先级任务阻塞是必定的，但被中优先级任务阻塞则是很无奈的。为了避免因为中优先级任务挟持低优先级任务从而阻塞高优先级任务的现象，可以采用一些必要的算法。

有两种经典的防止优先级反转的算法：

- 优先级继承策略（Priority inheritance）：当一个任务占有了资源并且随后阻塞了其他申请该资源的任务时，该任务将临时改变它的优先级为所有申请该资源的任务中的最高优先级，并以这个临时优先级在临界区执行。当任务释放资源后，则恢复它原有的优先级。从行为上看，占有资源的任务的优先级将是“水涨船高”式的多次改变，因为它的优先级最高，所以它不会被曾经比它优先级高的那些任务抢占。操作系统从优先级角度安排它尽快执行，尽快释放资源，但是这样做操作系统却牺牲了中等优先级任务的调度机会。
- 优先级天花板策略（Priority ceilings）：将申请（占有）资源的任务的优先级提升到可能访问该资源的所有任务的最高优先级（这个最高优先级称为该资源的优先级天花板）。

1. 优先级继承策略

优先级继承策略如图 1-18 所示。

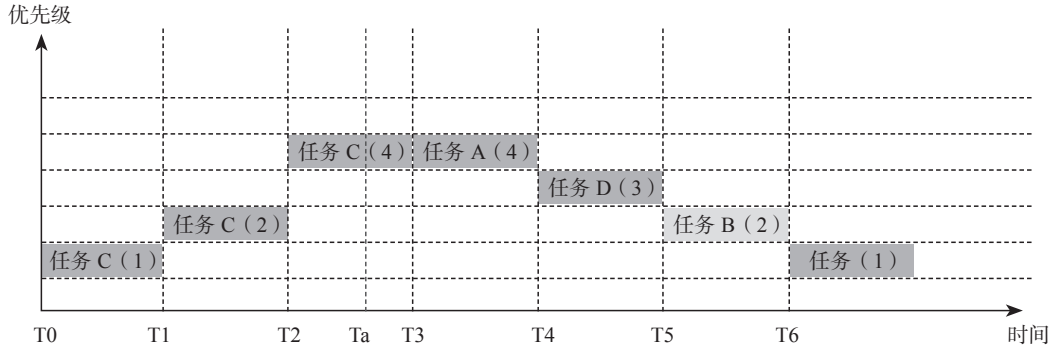

图 1-18 优先级继承策略

⊖ 优先级反转是由来已久的问题，自从出现多任务系统后，就一直困扰着开发人员，很典型的一个优先级反转引起的案例是在 1997 年 7 月美国探路者火星车（Pathfinder）发生的在火星表面不停重启的故障，读者如果有兴趣可以自己去查阅相关资料。

- T0 时刻，只有任务 C 处于运行状态，在运行过程中，任务 C 得到共享资源 R。
- T1 时刻，任务 B 抢占任务 C，并尝试获得资源 R，因为优先级继承的原因，任务 C 的优先级被提升到任务 B 的优先级；任务 B 被阻塞。
- T2 时刻，任务 A 抢占任务 C，并尝试获得资源 R，因为优先级继承的原因，任务 C 的优先级被提升到任务 A 的优先级；任务 A 被阻塞。
- 在 Ta 时刻，任务 D 就绪，但因为此时任务 C 的优先级已经被提升并且比任务 D 优先级高，所以任务 D 不能抢占任务 C，任务 C 继续运行。
- T3 时刻，任务 C 释放资源，它的优先级恢复到原有优先级。任务 A 得到资源，并因为优先级原因抢占任务 C。
- T4 时刻，任务 A 释放资源，结束运行。任务 B 得到资源。但此时因为任务 D 优先级高于任务 B，所以任务 D 开始运行。
- T5 时刻，任务 D 结束运行，任务 B 开始执行。

2. 优先级天花板策略

优先级天花板策略如图 1-19 所示。

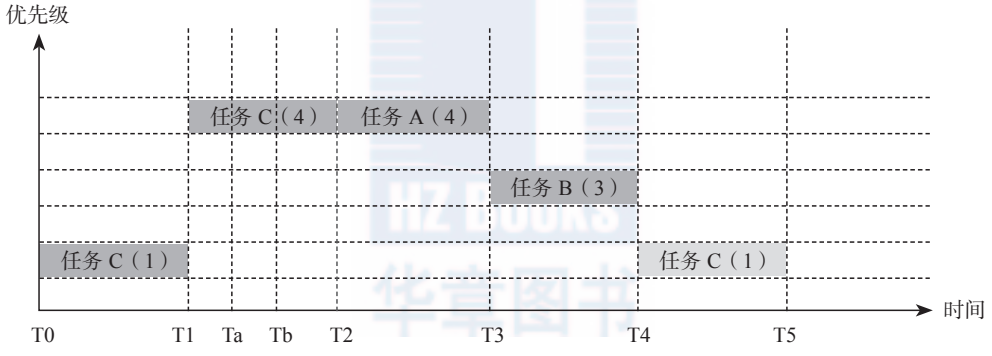

图 1-19 优先级天花板策略

- T0 时刻，只有任务 C 处于运行状态。
- T1 时刻，任务 C 得到共享资源 R，因为优先级天花板策略的原因，任务 C 的优先级提升到全部可能访问该资源的任务的最高优先级。
- Ta 时刻，任务 A 抢占任务 C 执行，随后尝试获得资源 R，但是失败并阻塞。任务 C 继续运行。
- Tb 时刻，任务 B 就绪，但是因为任务 C 优先级更高，所以只能等待执行。
- T2 时刻任务 C 释放资源 R，任务 A 得到资源 R。因为使用优先级天花板策略，任务 C 优先级恢复到原有优先级。任务 A 抢占任务 C 开始运行。
- T3 时刻，任务 A 结束运行。任务 B 开始运行。
- T4 时刻，任务 B 结束运行。任务 C 开始继续运行。

优先级继承策略对任务执行流程的影响相对较小，因为只有当高优先级任务申请已被低

优先级任务占有的共享资源这一事实发生时，才提升低优先级任务的优先级。而天花板策略是谁占有就直接升到最高。形象地说，优先级继承策略是“水涨船高”，而优先级天花板策略则是“一次到位”。

1.4 中断机制

中断机制是处理器的重要基础设施，用来应对各种事件的响应和处理。当外设或者处理器自身有事件发生时，处理器会暂停执行当前的代码，并转向处理这些中断事务。在处理器与外设间的交互大多采用中断来完成，中断系统能极大提高系统的效率。

发出中断请求的来源叫作中断源。根据中断源的不同，可以把中断分为以下三类：

1. 外部中断

外部中断是指由系统外设发出的中断请求，如串口数据的接收、键盘的敲击、打印机中断、定时器时间到达等。外部中断大多是可以屏蔽的，程序可以根据具体需要，通过中断控制器来屏蔽这些中断请求。

2. 内部中断

内部中断指因处理器自身的原因引起的异常事件，如非法指令、总线错误（取指）或者运算出错 (除 0) 等。内部中断基本是不可屏蔽的中断。

3. 软件中断

软件中断是一种特殊的中断，它是程序通过软件指令触发的，从而主动引起程序流程的变化。比如在用户级运行的程序在某时刻需要访问处理器中受到保护的寄存器，则可以通过软件中断进入系统级，实现权限的提升。

在不同的处理器中，以上三种中断可能有不同的名称或者概念，但从技术层面划分，各种中断基本都属于这三类。所以读者不必纠结于不同处理器上的具体称谓。比如在 ARM Cortex-M3 内核中，将软件中断和内部中断统称为异常，把外部中断称为中断，并通过中断向量表把这些中断和异常组织在一起，如表 1-2 所示。

表 1-2 中断向量表

编号	类型	介绍
0	N/A	没有异常
1	复位	系统复位
2	NMI	不可屏蔽中断
3	硬 fault	所有不被处理的 fault 都统一引起硬 fault
4	内存管理 fault	
5	总线 fault	
6	用法 fault	
7-10	保留	
11	SVC Call	由系统调用指令 SVC 引起

（续）

编号	类型	介绍
12	调试器	
13	保留	
14	PendSV	系统级中断服务
15	SysTick	系统时钟中断
16	IRQ#0	0 号外中断
17	IRQ#1	1 号外中断
…		
255	IRQ#255	255 号外中断

Cortex-M3 在内核中拥有一个异常处理系统，用来支持各种系统异常和外部中断。其中编号为 1 ～ 15 的对应系统异常，其余则是外部中断。

1.4.1 中断流程概述

不同处理器上的中断处理流程大致是相同的，但也有些细微差别。所以这里我们只介绍中断大概的几个阶段，如图 1-20 所示。

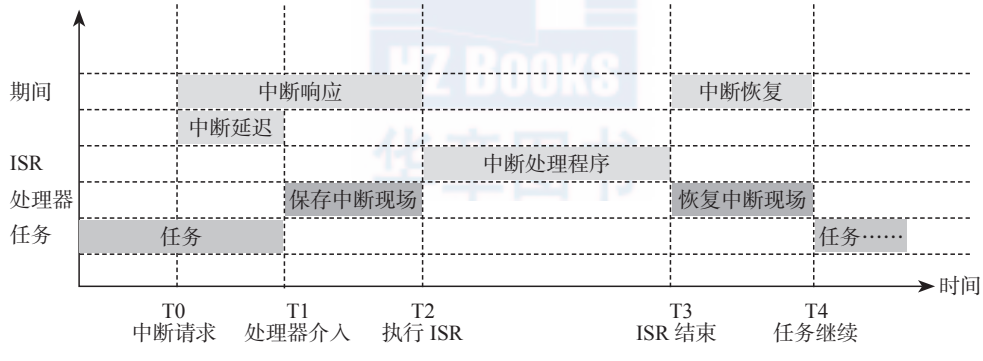

图 1-20 中断的几个阶段

- T0 时刻，用户程序正在执行，此时有外部设备产生中断请求。
- T1 时刻，处理器开始处理外部中断，保存中断现场。
- T2 时刻，处理器开始执行中断处理器函数。
- T3 时刻，用户中断处理函数结束，处理器开始恢复中断现场。
- T4 时刻，被中断的用户程序继续执行。

其中从 T0 时刻中断产生到 T1 时刻处理器开始处理中断的这段时间称为中断延时，这个和处理器是相关的。也就是说，外部中断并不是一发生就被立刻处理的。

从 T1 时刻开始，处理器开始处理中断，一般来说，此时最重要的工作就是保留中断现场，以保证能够正确恢复任务的执行。中断现场主要的内容就是处理器上下文，而数据一般是保存到某个具体的栈中。这个同样与处理器相关。

T2 时刻处理器就可以正常去执行用户定义的中断处理函数。而从 T0 到 T2 这段时间被称为中断响应时间。

T3 时刻中断处理函数运行完毕，通过特殊的指令或者流程，处理器从栈中开始恢复本次中断的现场。T3 到 T4 的时间称为中断恢复时间。

以上是对流程最简单的介绍，其实中断流程还有很多细节问题，比如中断嵌套的问题，处理器保存上下文的问题，RTOS 内核介入中断处理的问题。

1.4.2 中断优先级

当几个中断同时产生时，首先响应哪个中断是个值得考虑的问题。当前绝大多数的处理器都支持中断优先级的概念，也就是说，为不同的中断源配置不同的优先级，优先级是固定的或者可以通过软件配置。当多个中断同时产生（或者说需要处理）时，首先响应优先级高的中断，这样就能优先处理高优先级的事件。不同的处理器可能有不同的中断优先级策略。

1.4.3 中断嵌套

与中断相关的另一个重要的问题是中断嵌套。当处理器正在处理某个中断时，如果有其他中断发生，那么就得仔细考虑如何处理新的中断。对于简单的处理器来说，可能本身并不支持中断嵌套，在中断响应阶段就把中断给关闭了。而当前大多数的处理器是支持中断嵌套的，方案基本是结合中断优先级的嵌套方式，原则是：

- ❏ 高优先级中断可以抢占低优先级中断。
- ❏ 同级中断不可抢占（包括自身）。
- ❏ 不能被立刻响应的中断会被悬挂 (Pending)，等待高优先级中断退出后才执行。

中断嵌套行为的如图 1-21 所示。

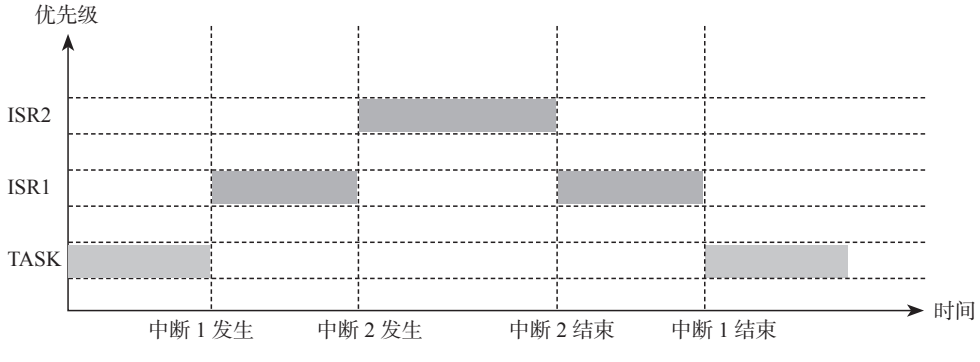

图 1-21 中断嵌套行为

1.4.4 中断时序

在前面我们介绍了中断的基本流程，结合前面章节介绍的嵌入式软件结构和实时操作系统的抢占方式，我们来具体分析一下中断时序的问题。

首先是前后台系统的中断时序，如图 1-22 所示。

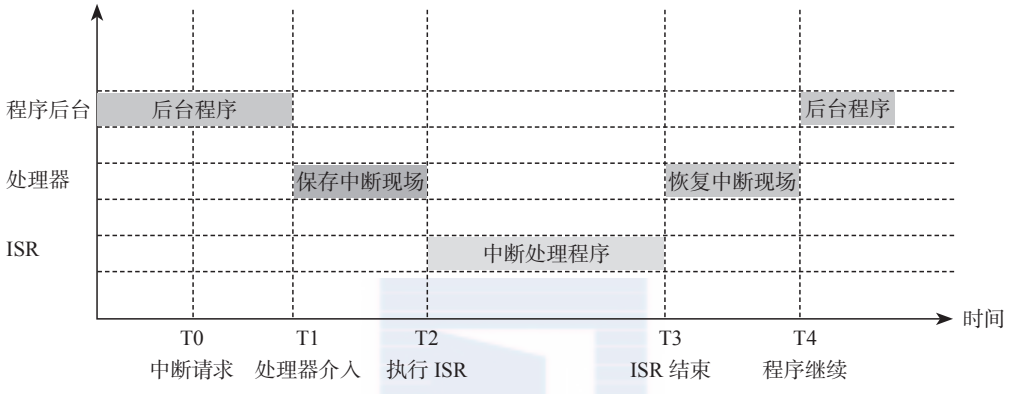

图 1-22 前后台系统的中断时序

可以看出前后台系统的中断时序很简单，这也说明了前后台的特点：实时响应事件、轮询处理事务。

图 1-23 演示了不可抢占式操作系统的中断处理流程。

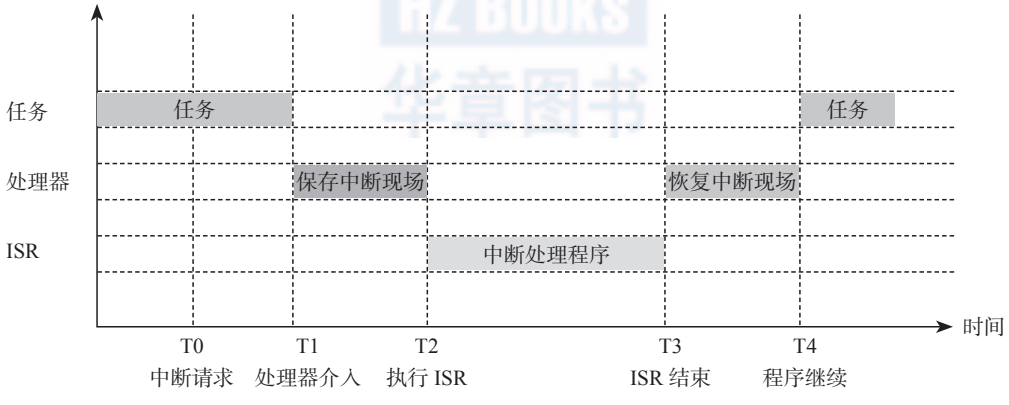

图 1-23 不可抢占式操作系统的中断处理流程

可以看出不可抢占式多任务系统的中断时序和前后台系统的中断时序很相似，需要注意的是前后台系统只有一个后台任务，所有中断打断的都是这个唯一的任务。而不可抢占式多任务系统则有很多任务，每个任务都可能被中断打断。

图 1-24 演示了可抢占式操作系统的中断处理流程。

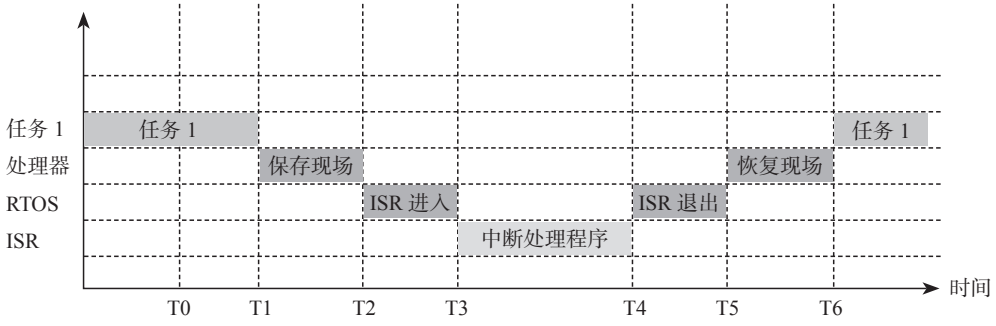

图 1-24 可抢占式操作系统的中断处理流程

如图 1-24 所示：

- T0 时刻，用户任务正在执行，此时有外部设备产生中断请求。
- T1 时刻，处理器开始处理外部中断，保存中断现场。
- T2 时刻，RTOS 介入，进行中断预处理，记录当前中断嵌套深度，查找用户登记的中断处理程序。
- T3 时刻，处理器开始执行中断处理器函数。
- T4 时刻，RTOS 介入，进行中断退出处理，如果当前中断是所有嵌套中断的最后一个，则进行任务调度处理，此时可能发生任务切换的准备。
- T5 时刻，处理器开始恢复中断现场，此时恢复的现场未必是 T1 时刻保存的任务现场，而是在 T4 时刻选择的最高就绪优先级的任务。
- T6 时刻，如果没有进行任务调度和切换，则被中断的用户任务继续执行，如图 1-24 所示。如果进行了任务切换，则如图 1-25 所示。

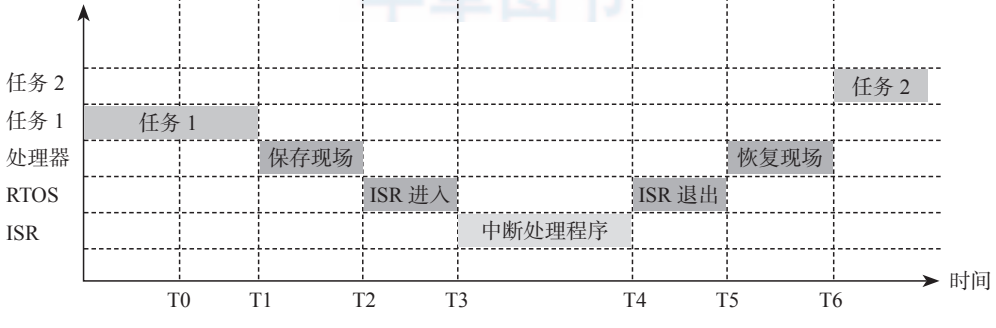

图 1-25 进行任务切换

这里需要解释一下为什么 RTOS 介入可抢占式操作系统的中断流程。因为在可抢占式操作系统中，当从中断返回时会处理任务抢占的问题，所以在每次中断进入时都要对中断嵌套计数做加法；在每次中断退出时都要对中断嵌套计数做减法，然后再继续检查中断嵌套计数是否为 0，如果是，则说明中断彻底退出了，需要进行任务调度的处理。这些操作都是由

RTOS 完成的。而不可抢占式操作系统则没有这些事情要做，所以不必对中断流程做特殊处理，基本和前后台系统的中断流程一致。

1.5 Trochili RTOS 介绍

Trochili RTOS 是一个全新的适用于嵌入式领域的实时操作系统，主要用 C 语言开发，支持多任务、多优先级、抢占式调度。英文名称 TROCHILI 取善鸣的小鸟之意，意味着体积小巧、动作灵敏。其主要特点如下：

- ❑ 支持抢占式调度多任务模型。
- ❑ 最多支持 32 个任务优先级，多个任务可以拥有同优先级。
- ❑ 不同优先级任务采用优先级调度，相同优先级任务间采用时间片调度。
- ❑ 支持用户回调定时器和任务定时器。操作系统内置用户定时器守护线程。
- ❑ 支持常见 IPC 机制，如 semaphore、mailbox、message、mutex、flag。
- ❑ 充分总结各种机制的共性和特性，基于通用 IPC 控制结构和操作流程，做了完整简洁的实现。
- ❑ 可配置的 IPC 调度机制，支持 FIFO 和优先级两种方式的线程阻塞队列。紧急消息操作优先普通消息操作。
- ❑ 大量 API 支持在 ISR 中调用。
- ❑ 代码实现简洁，注释完备，有十分详尽的中文注释。

Trochili RTOS 基本实现了上面介绍的 RTOS 的知识点，并且有自己的独特实现。目前还不能说它是一套成熟稳定的商业代码，但作为学习和理解 RTOS 确实是不可多得的好资料。从代码、注释和文档上看，都提供了大量的图表和大段的中文注释，这些是在别的 RTOS 代码中看不到。

Trochili RTOS 主要实现了以下几个功能：

- ❑ 线程管理和调度
- ❑ 信号量和互斥量
- ❑ 邮箱和消息队列
- ❑ 事件集合
- ❑ 定时器和其守护线程
- ❑ RTOS 移植和启动
- ❑ 调试选项

作者将按照原理、设计、实现和应用的思路在随后的章节中逐步介绍 Trochili RTOS 的各个功能模块。从多个角度向读者充分展示 RTOS 的核心和细节。希望读者不仅能从中理解 RTOS 的概念和使用，还能实现自己的想法和改进方案。相信下面各章的内容一定不会让你失望。另外因为现在 Trochili RTOS 只是实现了内核部分，所以下面章节常使用类似“Trochili RTOS”、“Trochili 内核”、“内核”这样的词，请读者不要疑惑。

第 2 章
线程管理与调度

在第 1 章中我们已经介绍了单核处理器上的多任务机制的基本知识。本章则主要介绍 RTOS 的多任务机制的设计和具体实现。首先介绍 Trochili RTOS 的多任务机制设计，然后通过流程图、代码演示的方式仔细讲解 Trochili RTOS 的多任务机制的实现。本章按照原理、设计、实现的思路，带领读者逐步深入了解和掌握 RTOS 的多任务机制。

2.1 线程结构设计

Trochili RTOS 目前的版本没有明确提供进程机制，而是以线程来代表用户任务，线程同时作为资源分配和任务调度的基本单位。从整体上考虑，可以把操作系统和用户任务看作是一个大的进程，而整个系统内则有多个用户线程。我们可以把 Trochili RTOS 定义为基于轻量级多线程的多任务系统。Trochili RTOS 的线程模型可以简要归纳为：

- ❏ 线程是任务代码在处理器上执行的过程
- ❏ 它是操作系统调度的基本单位，也是资源分配的基本单位
- ❏ 线程对象由线程结构、线程栈、线程函数和相关的线程数据等部分组成
- ❏ 线程具有多种运行状态
- ❏ 线程的状态及其所处的线程队列是匹配的

2.1.1 线程的结构设计

Trochili RTOS 是通过线程结构管理线程的，每个线程都存在一个线程结构。线程结构定义在文件 include\kernel\thread.h 中，其中包括操作系统用来管理线程的全部信息。作为操作系统的一种核心数据结构，线程结构决定了操作系统的很多特点。Trochili RTOS 当前版本的线程结构定义如下。

代码清单2-1：线程结构定义

```
1.  /* 操作系统线程结构定义，用于保存线程的基本信息 */
2.  typedef struct ThreadDef
3.  {
4.      TThreadID ThreadID;             /* 线程 ID                    */
```

```
5.      TStackAddr StackBase;           /* 线程堆栈栈底指针                */
6.      TStackAddr StackTop;            /* 线程堆栈栈顶指针                */
7.      TThreadStatus Status;           /* 线程状态                        */
8.      TPriority Priority;             /* 线程当前优先级                  */
9.      TPriority BasePriority;         /* 线程基本优先级                  */
10.     TTimerTick Ticks;               /* 时间片中还剩下的 ticks 数目     */
11.     TTimerTick BaseTicks;           /* 时间片长度（ticks 数目）        */
12.     TTimerTick TotalTicks;          /* 时间片长度（ticks 数目）        */
13.     TThreadEntry Entry;             /* 线程主程序地址                  */
14.     void* EntryArg;                 /* 线程主程序参数                  */
15.     TThreadQueue* Queue;            /* 线程所属线程队列指针            */
16.
17. #if (TCL_IPC_ENABLE)
18.     TIpcRecord IpcRecord;           /* 线程互斥、同步、通信的控制结构  */
19. #endif
20.
21. #if (TCL_TIMER_ENABLE)
22.     TTimer Timer;                   /* 线程自带定时器                  */
23. #endif
24.
25. #if (TCL_THREAD_AUTHORITY_ENABLE)
26.     TAuthority Authority;           /* 线程管理的授权使能              */
27. #endif
28.
29.     TLinkNode LinkNode;             /* 用来组成各种线程队列的节点      */
30. } TThread;
```

- ThreadID 线程编号：每个线程在初始化时都由操作系统自动分配一个 ID。
- StackBase 线程堆栈栈底指针：在初始化线程的时候需要用户提供一个数组作为线程的栈，数组的地址 + 数组长度的值会被作为线程栈起始地址（栈向下增长的情况）。
- StackTop 线程堆栈栈顶指针：在线程初始化的时候被设置，根据处理器对线程栈的要求有所不同。注意，当线程执行时，虽然线程的栈顶随函数调用而变化，但改变的只是处理器的相关寄存器，而不会更新本成员，只有在线程切换时才会更新本成员。
- Status 线程状态：线程状态在下文将专门介绍。
- Priority 线程当前优先级：因为操作系统需要支持互斥量的优先级继承策略，所以线程运行时的优先级和线程的基本优先级是分别保存的。本成员保存的是线程运行时的优先级。在没有互斥量的影响时，它的数值和线程基本优先级是相同的。
- BasePriority 线程基本优先级 ：操作系统支持线程动态优先级，这里保存的是线程实际的优先级，可以被相关线程管理函数修改。
- Ticks 时间片中剩余时钟节拍数：操作系统支持线程时间片机制，不同线程可以拥有不同长度的时间片。时间片长度以系统时间节拍为基数。本成员保存的是线程当前时间片所剩余的数目，一旦减少到 0 就要考虑时间片调度了。
- BaseTicks 时间片长度：线程时间片的长度，当发生时间片调度时，用来恢复线程当

前时间片计数。操作系统支持对时间片长度的修改，用户可以通过 API 来改变线程时间片的长度。当修改当前线程的时间片长度时，不必等待下次调度，可以立刻生效。

- ❏ Entry 线程函数入口：在当前操作系统中，线程函数的主体必须是一个无限循环的函数。
- ❏ EntryArg 线程函数参数：作为一个指针，可以指向任何类型的数据。
- ❏ Queue 线程所属线程队列指针：内核中有多种类型的线程队列，下面章节会详细介绍。非阻塞状态的线程所属的队列由本成员保存。
- ❏ IpcRecord 线程互斥、同步、通信的控制结构：线程因 IPC 而阻塞时，需要记录下很多信息，包括阻塞在哪个 IPC 对象上，需要传递的数据地址等。这些阻塞信息都存储在这个成员里。(有些内核会把这个结构放在线程 / 进程栈中)。
- ❏ Timer 线程自带定时器：用来实现线程延时和在 IPC 对象上的时限阻塞机制。
- ❏ Authority 线程操作的授权使能：每个线程都可以对各种线程管理功能做出不同的配置。假如某线程不希望被挂起，那么在对本成员做出相应的配置后，所有对它的挂起操作都不会成功。
- ❏ LinkNode 线程队列节点：用来保存线程所属队列的实际节点结构，通过本成员可以把线程组织成各种链表结构。
- ❏ TotalTicks 内核运行时间计数：保存线程实际运行的时间长度，当每次时间片中断时，更新一次当前线程的运行时间计数。

图 2-1 演示了一个线程结构及其基本存储空间的分布。

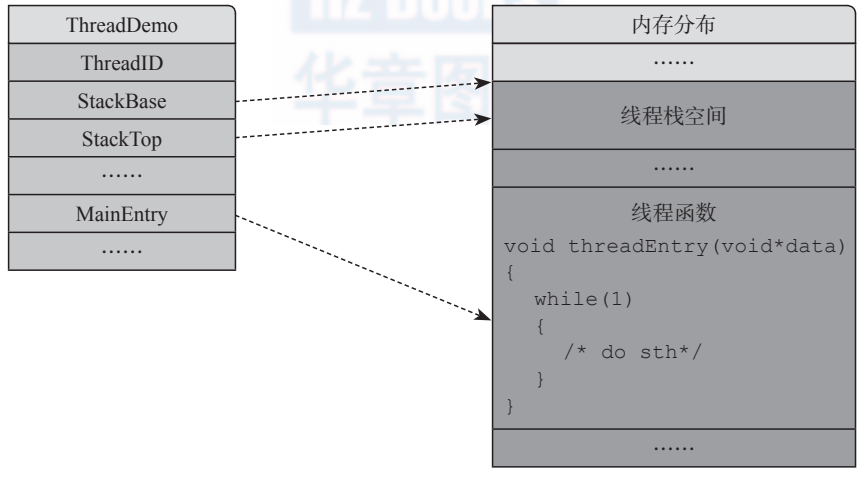

图 2-1　线程结构演示

2.1.2　线程的状态

在实际运行时，线程可能会处于不同的状态。Trochili RTOS 中的线程主要包括 5 种状态。

代码清单2-2：线程状态定义

```
1. /* 线程状态定义 */
2. typedef enum
3. {
4.     eThreadDormant = 0,          /* 休眠态       */
5.     eThreadReady,                /* 就绪态       */
6.     eThreadBlocked,              /* 阻塞态       */
7.     eThreadDelayed,              /* 延时态       */
8.     eThreadSuspended,            /* 挂起态       */
9. } TThreadStatus;
```

这些状态的解释如下。

- 休眠态：处于此状态的线程不再参与线程调度。线程初始化后即处于此状态。
- 就绪态：代表线程正在参与内核调度，随时可能被内核加载到处理器中运行。当前内核并没有单独设置一个运行态来代表正在处理器上运行的线程，而是以一个内核全局变量“当前线程”来代替。
- 阻塞态：说明线程因为某些条件不允许而无法完成一些具体操作，只好暂停运行，等待条件满足后继续运行。
- 延时态：线程因为在时间方面的要求，暂停运行，延时一段时间继续运行。
- 挂起态：处于此状态的线程不再参与线程调度，直到解除挂起为止。

线程在执行过程中，状态是经常变化的。线程的状态迁移如图 2-2 所示。

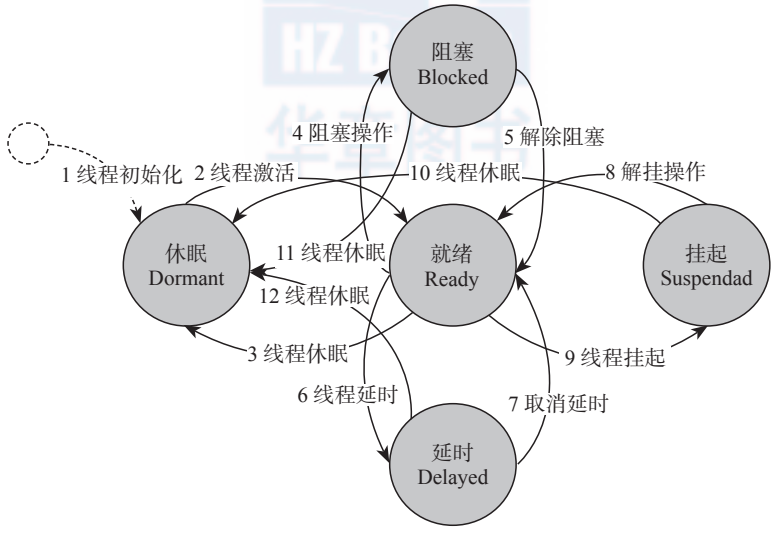

图 2-2　线程状态迁移图

线程的状态迁移和它的行为或对它的操作是相关的。对图 2-2 的分析如下：

（1）线程初始化操作

线程被初始化后就处于休眠状态，置于内核线程辅助队列中。此时它并不参与线程调

度，只是安静地待在队列中等待内核激活。

（2）线程激活操作

通过线程激活操作将线程转为就绪状态，并置于内核线程就绪队列中，等待内核调度。在内核中某一时刻可能多个线程处于就绪状态，但是只有一个线程在运行，称为当前线程。当前线程保持在内核线程就绪队列中。

（3）线程休眠操作

通过线程休眠操作可以将处于任何状态的线程（包括当前线程）返回到休眠状态。这意味着被休眠的线程不再参与线程调度，直到再次激活。

（4）线程阻塞操作

如果线程尝试访问某些资源但是由于某些原因而没能成功，根据访问规则，如果是非阻塞的访问方式，那么线程返回失败结果后继续运行；如果是阻塞的访问方式，那么线程就会被内核阻塞在该资源的线程阻塞队列上（每个拥有线程阻塞特性的资源都存在相关的线程阻塞队列，用来保存那些希望得到资源但又不能得到资源的线程）。随后，内核发起线程调度，使得当前线程放弃处理器，内核会加载其他线程来运行。

（5）线程阻塞解除操作

在线程释放资源时，假如该资源的线程阻塞队列上有线程被阻塞，那么当前线程会通过内核从这些被阻塞的线程中唤醒一个线程，并使它得到该资源，随后内核将该线程加入内核中的线程就绪队列。（因为有新的线程可以运行，所以此时内核尝试发起线程调度。但是这样也只是给了新线程一个参与竞争处理器资源的机会，不保证新线程能够立刻得到处理器。）

（6）线程延时操作

线程调用延时函数时，线程首先将被延时的线程置于延时状态并加入内核辅助线程队列，然后开启该线程的定时器。因为在等待定时结束的这段时间里，该线程不再参与内核线程调度，所以如果被延时的是当前线程则需要立刻进行一次线程调度。

（7）取消延时操作

包括线程的延时操作被明确终止，或者延时时间到达。前者会提前将线程从延时状态中恢复，继续参与线程调度。

（8）线程挂起操作

对处于就绪或者运行的线程进行挂起操作之后，就直接转为挂起状态，此时被挂起的线程会从线程就绪队列转到线程辅助队列。

（9）线程唤醒操作

处于挂起的线程可以被唤醒，线程从线程辅助队列中转移到线程就绪队列，并且状态设置为就绪状态，等待被处理器再次运行。

2.1.3 线程优先级

线程的优先级用来描述线程的重要性，它决定线程在获得内核调度机会上的能力。在线

程调度时，优先级高的线程会优先得到调度机会。Trochili RTOS 采用的是动态优先级机制，线程的优先级可以在运行时修改。线程优先级的范围是 0 ～ 31，数值越小说明优先级越高。多个线程可以拥有不同的优先级，也可以拥有相同的优先级。

用户可以通过系统调用直接改变某个线程的优先级，也可以通过其他操作间接地修改某个线程的优先级。当通过 API 函数来修改线程优先级时，需要注意某些特殊的内核线程可能没有被赋予“优先级可变”的授权，对这样的线程进行优先级的修改是不会成功的。

内核默认的线程优先级分配原则是：优先级 0 ～ 2 和 30 ～ 31 共 5 个优先级保留给内核线程使用，其余 27 个优先级分配给用户线程使用。内核 IDLE 线程使用最低优先级 31。如表 2-1 所示。

表 2-1 线程优先级分配

线程优先级	作用	备注
0		内核保留
1	内核定时器守护线程优先级	内核占用
2		内核保留
3 ～ 29	用户线程优先级可选范围	
30		内核保留
31	内核 IDLE 线程优先级	内核占用

2.1.4 线程时间片

在线程时间片长度的设置上，不同线程可以拥有不同长度的时间片，并且线程时间片长度也可以通过相关的 API 函数修改，即 Trochili RTOS 支持动态时间片分配方案。注意某些特殊的内核线程可能没有被赋予“时间片长度可改”的授权，对这样的线程进行时间片长度的修改是不会成功的。

用户可以通过时间片和优先级对系统中的线程做精细的配置。原则上，那些对响应时间要求高的线程需要配置为高优先级；需要处理大量数据的线程应该配置成长时间片。对这两个参数的配置可以组合出精细的系统行为。

2.1.5 线程栈管理

在 Trochili RTOS 线程管理过程中，应该非常注意线程栈的使用。Trochili RTOS 没有采用任何保护机制来确保用户内存的安全，线程和内核处于同一个地址空间，各个线程以及内核的数据和代码是统一编址的。

内核中的每个线程都有自己的栈空间。用户线程栈是在用户线程初始化时指定的，常见的方式是以全局数组作为线程栈。栈必须声明为整数类型（字长），并且必须由连续的内存空间组成。具体实现方式请参考线程初始化函数的实现。

在系统运行时，内核代码和用户线程一样，同样需要一个运行栈。Trochili RTOS 中并没有为内核代码特别设定运行栈，而是使用当前用户线程的运行栈。当用户线程调用内核函数

时，被调用的内核函数将使用当前用户线程的栈；但是内核中所有中断服务程序会共享一个独立的中断栈，这个中断栈与用户线程使用的运行栈没有任何联系。系统运行时，要保证任何类型的栈都不会溢出，否则内核会发生未知的混乱。

栈的管理跟具体处理器和编译器的特性相关。线程栈中保存的主要是线程执行时的局部变量和中断相关的数据，包括：

- ❏ 线程被中断时处理器自动保存的寄存器数据。
- ❏ 调用函数时，主调函数保存的寄存器数据。
- ❏ 函数被调用时，被调函数内部的局部变量。

2.1.6 线程函数和线程数据

线程函数必须是一个无限循环体，应用功能代码在循环体内，这样保证线程代码不会执行完毕而直接返回，否则会因内核无法捕获线程退出消息，从而导致内核错误。如果确实需要退出，则应该通过系统调用，显式通知内核该线程需要退出。线程函数类型如下：

```
typedef void (*TThreadEntry)(void* arg);
```

注意，参数是个线程函数的参数，它是一个 void* 类型的变量，这样在线程初始化的时候可以带入一个任意参数给线程作为初始化数据。

线程函数结构如代码清单 2-3 所示。

代码清单2-3：线程函数结构

```
static void ThreadEntry(void* pArg)
{
  while (1)
  {
    /* application code, do sth. */
  }
}
```

需要注意的是，在线程函数里不能长时间关闭中断。因为多任务需要保证用户任务能被中断打断，这样操作系统才能及时响应中断。

线程数据主要包括栈数据、全局数据和动态数据。栈中的数据编译器会确保完整和正确。但是全局数据是各个线程和内核都能访问的，所以对全局数据的访问必须是互斥的。

2.2 线程队列设计

内核中的线程结构不是分散在内核各处游离存在的，每个线程在某个时刻必定属于某个线程队列。线程队列就是内核用于管理线程的数据结构，它基本上是按照线程状态的不同而设置的。线程的状态和它所处的线程队列是匹配的。在当前版本的内核中，主要包括的 3 种线程队列如表 2-2 所示。

表 2-2 线程队列

线程队列	作用	个数
线程就绪队列	用于保存处于就绪状态的线程结构，包括当前线程（隐含运行状态）	1
线程辅助队列	用于保存处于挂起、休眠、延时状态的线程结构	1
线程阻塞队列	用于保存阻塞在 IPC 对象上的线程结构	视 IPC 对象个数而定

需要注意的是，处于运行状态的线程其实属于就绪队列。前两个线程队列属于第一种线程队列结构，这种线程队列结构的特点是能够实现固定时间的调度算法。内核中只有一个线程就绪队列和一个线程辅助队列。

在 IPC 机制中的各种对象类型，大多具有线程阻塞队列，这种队列和线程就绪队列及辅助队列的结构是不一样的。并且这种类型的队列的个数是不确定的，是根据实际使用的 IPC 对象的多少决定的。本章只涉及第一种线程队列类型。

基本线程队列结构定义在文件 include\kernel\thread.h 中，具体内容如代码清单 2-4 所示。

代码清单2-4：基本线程队列结构定义

```
/* 线程队列结构定义，该结构大小随内核支持的优先级范围而变化，可以实现固定时间的线程调度算法 */
typedef struct ThreadQueueDef
{
    TPriorityMask PriorityMask;                 /* 优先级掩码 */
    TLinkNode* Handle[THREAD_PRIORITY_NUM];     /* 线程分队列 */
} TThreadQueue;
```

1. 线程队列结构分析

- PriorityMask 优先级就绪标记：每个 bit 代表了相应优先级的线程队列的情况。如果某个优先级的分队列中存在线程，则这个成员相应的 bit 位会被置位。
- Handle[ ] 线程优先级分队列：该数组中的每个元素代表了一个优先级分队列的指针。相同优先级的线程处在同一个优先级分队列中。

线程队列结构如图 2-3 所示。

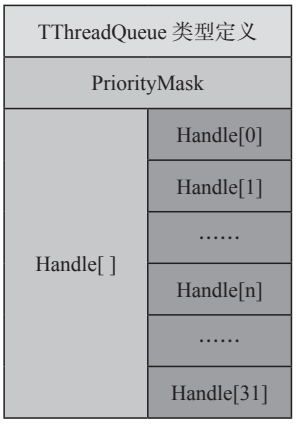

图 2-3 线程队列结构

从上面的线程队列定义可以看出，线程队列内设置了线程分队列的指针数组，这些分队列指针指向了同优先级线程组成的双向循环链表。

成员 PriorityMask 表明了哪些优先级的线程处于队列中。通过快速的计算算法，内核可以在固定时间内找到线程队列中最高的就绪优先级，然后找到处于该优先级队列的第一个线程。

2. 线程队列操作

和线程队列相关的操作主要包括线程加入队列、线程移出队列、选择队列中线程的最高优先级和选择队列中的最高优先级线程。这其中主要涉及链表的操作和查找优先级算法。线程队列内，按照优先级的不同有多个双向循环链表。线程出入某个线程

队列时，就是对这些链表进行的操作，同时完成线程队列其他数据的更新。

- ❑ 线程加入队列时，根据优先级查找和它匹配的链表头，将它追加到链表中，最后更新线程所属优先级是否有线程存在的标记。
- ❑ 线程从队列离开时，同样根据优先级查找和它匹配的链表头，将它从链表中删除，最后更新线程所属优先级是否有线程存在的标记。

3．线程队列图示

某个时刻内核中线程就绪队列和线程辅助队列如图 2-4 和图 2-5 所示。

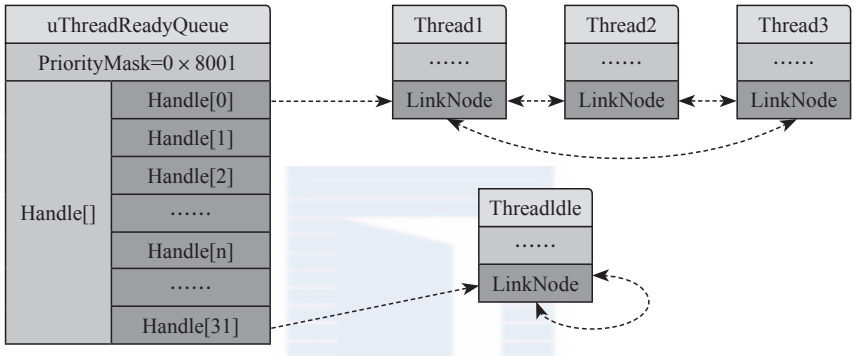

图 2-4　线程就绪队列图示

从图 2-4 可以看出，在当前的系统中，线程 Idle、线程 1、线程 2、线程 3 就绪。并且进一步可知，线程 1 正在运行中。线程就绪队列的优先级掩码为 0x8001，即优先级 0 和优先级 31 有线程就绪。

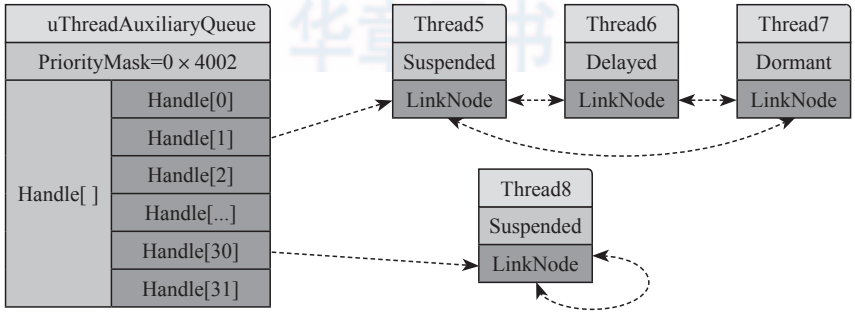

图 2-5　线程辅助队列图示

图 2-5 说明，在当前的系统中，线程 5 和线程 8 被挂起，线程 6 被延时，线程 7 则处于休眠状态。线程辅助队列的优先级掩码为 0x4002，说明优先级 1 和优先级 30 的队列中有线程存在。

到此为止，我们已经介绍了 Trochili RTOS 的多任务操作系统模型，包括线程和线程队列的各个细节问题。下面我们就来看看如何实现这个多任务操作系统模型。

2.3 线程调度机制设计

Trochili RTOS 的线程调度机制包括优先级抢占机制和时间片轮转机制。在不同优先级的线程之间采用优先级抢占机制；在相同优先级线程之间采用时间片轮转机制。这种调度算法可以兼顾线程在优先级上的不同和相同优先级线程之间在占有 CPU 机会上的公平性。支持这种调度机制的核心数据结构就是前面介绍的线程队列结构。

另外，内核进行任务调度操作所花费的时间是常数（IPC 线程队列不是这样的），与应用程序中建立的任务数无关，这归功于内核队列结构的定义和队列操作函数的实现。线程的优先级和时间片是在建立的时候分配的，并且可以在运行期间修改。内核对线程优先级和时间片管理可以做到非常灵活。

2.3.1 线程调度模型

在内核中的某一时刻，可能会有多个线程同时处于就绪状态，内核必须决定某个时刻应该执行哪个线程，调度算法是内核的关键特性。在线程调度部分，需要注意有几个可能发生线程调度的时机和可能引起内核调度的函数。清楚内核会在什么情况下发生调度是很重要的，这里我们总结一下 Trochili RTOS 的调度时机：

- ❏ 当前线程直接放弃处理器，通过 API 函数实现。
- ❏ 当前线程间接放弃处理器（无法获得资源、优先级变化、线程挂起、线程休眠、线程阻塞、线程延时等操作）。
- ❏ 时钟节拍中断处理过程中，当前线程时间片用尽，被迫放弃处理器。
- ❏ 中断返回过程中，当前线程被更高就绪优先级线程抢占。

在上面的介绍中，我们把内核调度分为主动调度方式和被动调度方式，同时线程调度也有直接和间接的区别。

从线程的状态、线程所处的线程队列、线程优先级、函数所处执行环境来考虑 Trochili RTOS 中的线程调度问题，可以总结为下面几点：

- ❏ 处于就绪状态的线程一定处于线程就绪队列。
- ❏ 处于阻塞状态的线程一定处于某个 IPC 对象的线程阻塞队列。
- ❏ 处于其他状态的线程一定处于内核线程辅助队列。
- ❏ 当前线程不一定处于内核线程就绪队列。
- ❏ 当前线程不一定处于内核线程就绪队列中最高优先级的分队列。
- ❏ 当前线程不一定处于内核线程就绪队列中最高优先级的分队列的队列头。
- ❏ 当前线程不一定是内核最高就绪优先级的线程。
- ❏ 当内核关闭线程调度时，当前线程不能离开内核线程就绪队列，但可以在就绪队列不同分队列中移动。
- ❏ 如果有线程加入内核线程就绪队列中，则可能需要进行线程调度。

❏ 当前线程从内核线程就绪队列中离开则必须申请线程调度。
❏ 在时钟节拍 ISR 中，需要处理当前线程的时间片和队列头指针的调整问题。
❏ 在时钟节拍 ISR 中，有时需要处理当前线程所在就绪线程队列的队列头指针问题。
❏ 只有处于线程就绪队列各分队列头位置的当前线程，才需要在时间片 ISR 中进行时间片调度。
❏ 在嵌套的中断 ISR 中，不必进行线程调度处理。

在实现内核管理函数时，均考虑了上面的因素，后续章节中将有更详细的解释。

2.3.2 线程调度算法

Trochili RTOS 的线程就绪优先级是放在内核线程就绪队列中的 PriorityMask 成员中的，该成员是一个 32 位的变量，每位代表了一个就绪的优先级，低位代表的优先级高于高位代表的优先级。在 Cortex M3 处理器上，可以采用两条特殊的指令 CLZ 和 RBIT 来快速查找处于就绪态的最高优先级任务。

❏ RBIT 把一个 32 位数水平旋转 180 度。
❏ CLZ 计算前导零的个数。

假设 PriorityMask 的二进制值为 00000000000000000000000000001100b，表示在优先级 2 和优先级 3 的线程队列中存在就绪线程。我们首先通过 RBIT 指令将 PriorityMask 数据变为 00110000000000000000000000000000b，然后通过 CLZ 计算出前导零的个数为 2，说明在最高就绪优先级的那个 BIT 前有两个 0，进而得知 BIT 2 就是当前最高就绪优先级。

在 Trochili RTOS 的线程就绪队列中，每个优先级的线程分队列占用 4 字节，32 个优先级占用共 128 字节（32×4）。出于对 RAM 空间的考虑，内核现在只设置了 32 个线程优先级，如果 RAM 许可，其实可以扩展更多优先级。

计算最高就绪优先级的算法不是唯一的，其他的 RTOS 在优先级调度算法上可能有更好的实现，读者可以自己研究一下。

2.3.3 线程调度步骤

内核的线程调度主要包含以下三个步骤：

❏ 处理当前线程的时间片、状态和线程所处的线程队列
❏ 根据调度算法查找新的就绪线程
❏ 在两个线程之间进行线程切换

其中线程切换是一个缜密的过程，核心功能需要由汇编代码来完成，因为 C/C++ 等高级语言是无法处理此时的处理器上下文的。内核首先把当前线程的寄存器上下文保存到它的线程栈中，更新线程结构中的线程栈顶成员。然后再把需要切换运行的线程的寄存器上下文从它的线程栈中加载到处理器寄存器组中，从而完成线程切换。

2.4 线程管理和调度实现

嵌入式操作系统的最基本功能就是任务的管理与调度。Trochili RTOS 提供基本的线程管理与调度 API 函数。

本章最后的内容主要分析了与线程管理与调度相关的各个 API 函数的功能与实现，对相关代码也进行了讲解，但是对于 API 函数内部调用的函数，则只是根据需要进行合理取舍后，对部分进行了分析讲解。在内核的源代码中有非常详细的注解，在本章中不再赘述。

Trochili RTOS 实现了如下线程管理和调度功能：

- ❏ 线程初始化（Init）
- ❏ 线程激活和休眠（Activate\DeActivate）
- ❏ 线程挂起和唤醒挂起（Suspend\Resume）
- ❏ 线程主动调度（Yield）
- ❏ 线程延时（Delay）
- ❏ 线程优先级管理（SetPriority）
- ❏ 线程时间片管理（SetTimeSlice）
- ❏ 线程管理授权（Autherity）

线程管理功能如表 2-3 所示。

表 2-3 线程管理功能

函数	功能	备注
TclInitThread	线程初始化	初始化线程结构和数据
TclSuspendThread	线程挂起	将就绪态的线程挂起
TclResumeThread	线程解挂	将挂起态的线程重新放回就绪队列
TclDelayThread	线程延时	将就绪态的线程延时一段时间
TclUndelayThread	线程延时取消	提前将延时的线程唤醒，解除延时
TclActivateThread	线程激活	将处于休眠状态的线程激活
TclDeActivateThread	线程休眠	将处于任何状态的线程置于休眠态
TclYieldThread	线程主动调度	运行中的线程主动放弃处理器
TclSetThreadPriority	修改线程优先级	修改任何状态的线程的优先级
TclSetThreadSlice	修改线程时间片	修改任何状态的线程的时间片长度

线程管理授权机制

Trochili RTOS 在设计的时候，在线程功能的使用方面做了特殊的处理，设计了一个线程管理操作的授权 / 验证机制。即对线程进行管理时，如果被操作的线程并没有使能相关功能，那么操作是不会成功的。这就给线程管理和内核的设计带来了很大方便，比如内核 IDLE 线程要一直处于就绪状态，不能被挂起、阻塞或者休眠。如果在每个任务管理的 API 中都判断被操作的线程是不是 IDLE 线程，那内核实现时就比较繁琐和凌乱了，特别是在内核内置的

线程较多时。而在 Trochili RTOS 中，则是通过检查 IDLE 线程的功能授权操作来完成的。每个线程都有它的操作授权集合，可以简单灵活地配置组合各种属性。比如内核 IDLE 线程只有被激活的授权，那么其他线程是无法通过 API 来使它休眠或者挂起的。

2.4.1 线程初始化

线程初始化函数为线程分配和初始化相关的数据结构信息。线程可以在多线程调度开始前初始化，也可以在其他线程的执行过程中初始化。它的主要操作有如下几个步骤：

- ❏ 构造线程原始堆栈栈帧
- ❏ 设置线程时间片相关参数
- ❏ 设置线程优先级
- ❏ 设置线程编号（ID）
- ❏ 设置线程定时器信息
- ❏ 设置线程链表节点信息
- ❏ 清空线程间通信信息
- ❏ 将线程加入内核辅助线程队列
- ❏ 设置线程状态为休眠态

初始化后的线程不会立刻执行，而是首先被置为休眠态，放在内核线程辅助队列（uThreadAuxiliaryQueue）中，然后在合适时机被内核或用户激活。激活后的线程处于内核线程就绪队列中，状态为就绪态。线程初始化函数本身没有复杂的逻辑，它按照参数来配置线程各结构成员。值得注意的是，在这个函数里会伪造线程首次运行时的栈帧，这个操作的实现和处理器硬件细节密切相关，将在第 10 章详细介绍。在这里只需要知道，线程首次运行时的栈帧是由内核伪造的就可以了。

2.4.2 线程激活

Trochili RTOS 实现了线程激活的功能。在前面介绍过，初始化后的线程都处于休眠状态，并不能立刻参与线程调度。需要通过线程激活 API 完成线程的激活后，线程才会进入内核就绪线程队列，参与系统实际的调度和运行。

只有处于休眠状态的线程才可以被线程和 ISR 激活。对线程进行激活时，会将处于休眠的线程从线程辅助队列 uThreadAuxiliaryQueue 中移出，随后放入线程就绪队列 uThreadReadyQueue 中。并且将线程的状态转为就绪态。某个线程被激活后，如果该线程的优先级高于内核当前最高就绪优先级，则内核立刻发出线程调度请求，完成线程的优先级调度抢占。

线程激活操作的核心步骤是：

- ❏ 把线程从内核辅助队列中移到内核就绪队列
- ❏ 设置线程状态为就绪

❑ 必要时，发起线程调度请求

整个线程激活的过程需要考虑以下几个问题：

❑ 线程是否允许进行激活操作

❑ 线程是否处于休眠态

❑ 当前内核是否允许线程调度

❑ 函数被线程调用还是被 ISR 调用

线程激活操作的主要流程如图 2-6 所示。

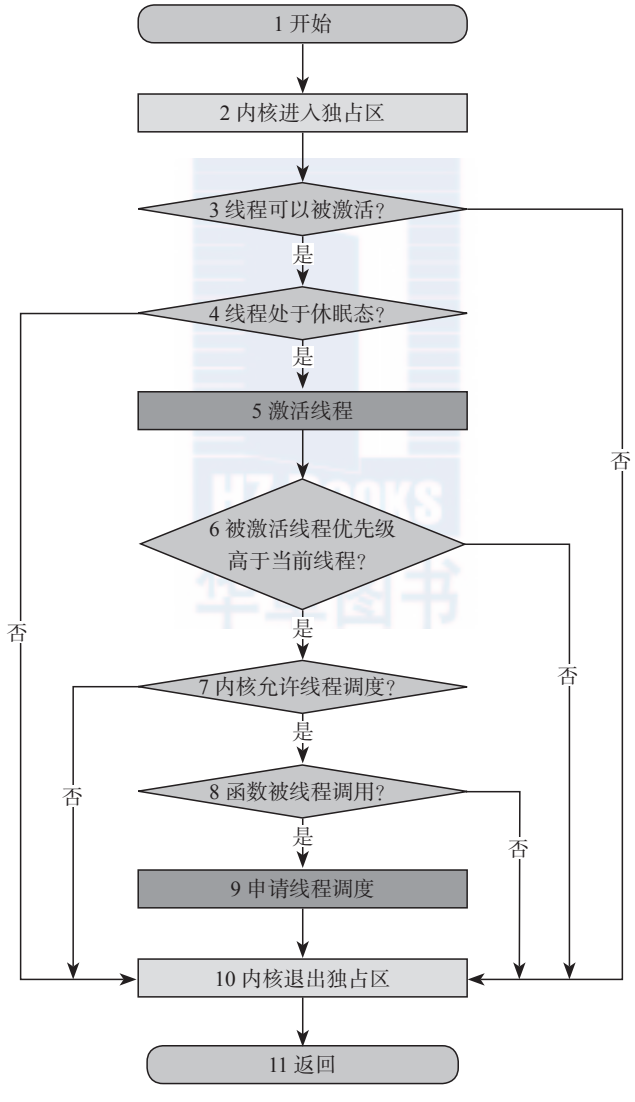

图 2-6 线程激活流程图

线程激活流程图分析：

STEP3 如果某个线程没有配置成可激活，那么对该线程的激活操作会失败。

STEP4 激活操作只对处于休眠状态的线程有效。

STEP5 激活线程，主要是对线程的队列和状态的操作。

STEP6 如果被激活的线程的优先级高于当前线程优先级则需要检查是否需要线程调度。

STEP7 如果内核没有关闭线程调度则需要考虑线程调度的问题。

STEP8 如果线程激活函数是被 ISR 调用则不需要发起线程调度。

STEP9 向内核发出线程调度请求。

STEP10 内核代码退出独占区。在中断退出时，内核可能响应线程调度请求。

线程激活的伪代码如代码清单 2-5 所示。

代码清单2-5：线程激活伪代码

```
1.  线程激活
2.  {
3.      关闭中断;
4.      如果线程可以被激活
5.      {
6.          如果线程处于休眠状态
7.          {
8.              将线程从内核辅助线程移出
9.              如果被激活线程是当前线程
10.             {
11.                线程在内核就绪线程队列的位置为队列头
12.             }
13.             将线程放入内核线程就绪队列
14.             设置线程状态为激活
15.
16.             如果被操作的线程的优先级高于线程就绪队列的最高优先级，
17.             并且内核此时并没有关闭线程调度，
18.             并且本函数被线程调度：
19.             {
20.                那么就需要发出线程调度请求
21.             }
22.         }
23.     }
24.     打开中断;
25. }
```

2.4.3 线程休眠

Trochili RTOS 实现了线程休眠的功能，可以将处于就绪状态、延时状态和阻塞状态的线程休眠，不再参与内核运行。如果休眠的是当前线程则需要进行线程调度。

线程休眠操作的核心步骤如下：

❑ 把处于阻塞状态的线程从 IPC 的线程阻塞队列中转移到内核线程辅助队列

❑ 把处于延时状态的线程保持在内核辅助队列线程队列
❑ 把处于挂起状态的线程保持在内核辅助队列线程队列
❑ 把处于就绪状态的线程从内核就绪线程队列中转移到内核线程辅助队列
❑ 设置线程状态为休眠
❑ 必要时向内核申请线程调度

整个线程休眠的过程需要考虑以下几个问题：

❑ 线程是否允许被休眠。
❑ 线程处于哪个状态？延时、就绪、挂起还是阻塞？
❑ 内核当前是否允许线程调度。
❑ 函数被线程调用还是被 ISR 调用。

线程休眠操作的主要流程如图 2-7 所示。

线程休眠流程图分析：

STEP3　如果某个线程没有配置成可休眠，那么对该线程的休眠操作会失败。

STEP4　线程处于阻塞态则调用相关函数将阻塞态的线程休眠。

STEP5　判断线程是否处于就绪态，如果是则处理就绪态的线程休眠。

STEP6　判断线程是否处于挂起态，如果是则处理挂起态的线程休眠。

STEP7　判断线程是否处于延时态，如果是则处理延时态的线程休眠。

STEP11　如果该线程是当前线程，则需要检查内核是否允许调度（这个判断条件包含了很复杂的情况）。

STEP12　如果内核不允许线程调度则直接退出。

STEP13　如果内核允许线程调度则将线程休眠。

STEP15　如果函数被线程调用则向内核申请线程调度。

STEP17　内核代码退出独占区后，可能发生线程切换。引起线程切换的原因有可能是线程休眠操作，也可能是其他原因。

线程休眠的伪代码如代码清单 2-6 所示。

代码清单2-6：线程休眠伪代码

```
1.  线程休眠
2.  {
3.      关闭中断;
4.      如果该线程可以被休眠
5.      {
6.          判断线程状态:
7.          {
8.              如果线程处于延时状态：将线程取消延时，但是线程仍然处于辅助队列，将线程设为休眠态
9.              如果线程处于阻塞状态：将该线程从阻塞队列中移出，加入线程辅助队列，将线程设为休眠态
10.             如果线程处于就绪状态:
11.             {
12.                 如果是当前线程
```

```
13.                 {
14.                     如果内核没有关闭线程调度
15.                     {
16.                         将该线程设为休眠态，并且加入线程辅助队列
17.                         如果是线程环境则需要马上开始线程调度
18.                     }
19.                 }
20.                 否则
21.                 {
22.                     将该线程设为休眠态，并且加入线程辅助队列
23.                 }
24.             }
25.             如果线程处于挂起状态：将该线程设为休眠态
26.         }
27.     }
28.     打开中断；
29. }
```

2.4.4 线程挂起

Trochili RTOS 实现了线程挂起的功能。处于就绪状态的线程可以被线程和 ISR 挂起，如果是当前线程被挂起则内核立刻发出线程调度请求；同样，只有处于挂起状态的线程才可以被线程和 ISR 解挂，如果解挂后的线程的优先级高于当前线程的优先级则内核立刻发出线程调度请求。

线程挂起操作的核心步骤是：

- ❏ 把线程从内核就绪队列中移到内核辅助队列
- ❏ 设置线程状态为挂起
- ❏ 必要时，发出线程调度

整个线程挂起的过程需要考虑以下几个问题：

- ❏ 线程是否允许被挂起
- ❏ 线程是否处于就绪态
- ❏ 内核是否允许线程调度
- ❏ 函数被线程调用还是被 ISR 调用

线程挂起操作的主要流程如图 2-8 所示。

线程挂起流程图分析：

STEP3　如果某个线程没有配置成可挂起，那么对该线程的挂起操作会失败。

STEP4　挂起操作只对处于就绪状态的线程有效。

STEP5　如果被挂起的线程是当前线程则说明挂起操作会引起线程调度。

STEP6　但如果此时内核不允许线程调度则本次挂起操作失败。

STEP7　处于就绪态的非当前线程直接进行线程挂起操作。

STEP8　挂起线程，主要是对线程队列和状态的操作。

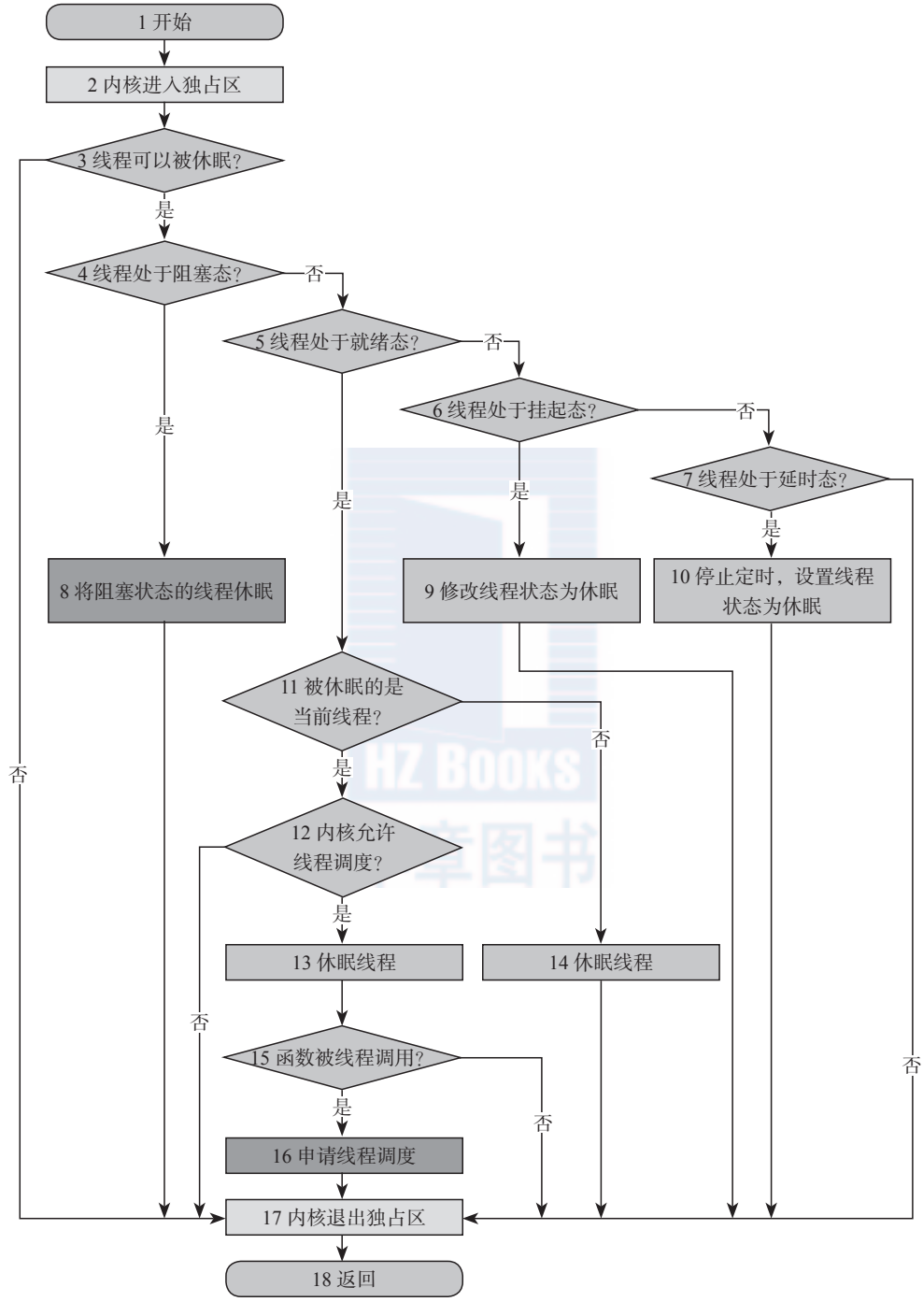
1 开始
2 内核进入独占区
3 线程可以被休眠?
是
否
4 线程处于阻塞态?
是
否
5 线程处于就绪态?
是
否
6 线程处于挂起态?
是
否
7 线程处于延时态?
是
否
8 将阻塞状态的线程休眠
9 修改线程状态为休眠
10 停止定时，设置线程状态为休眠
11 被休眠的是当前线程?
是
否
12 内核允许线程调度?
是
否
13 休眠线程
14 休眠线程
15 函数被线程调用?
是
否
16 申请线程调度
17 内核退出独占区
18 返回

图 2-7　线程休眠流程图

STEP9　检查函数是否被线程调用。

STEP10　如果是则向内核申请线程调度。

STEP11　内核代码退出独占区后，可能发生线程调度。

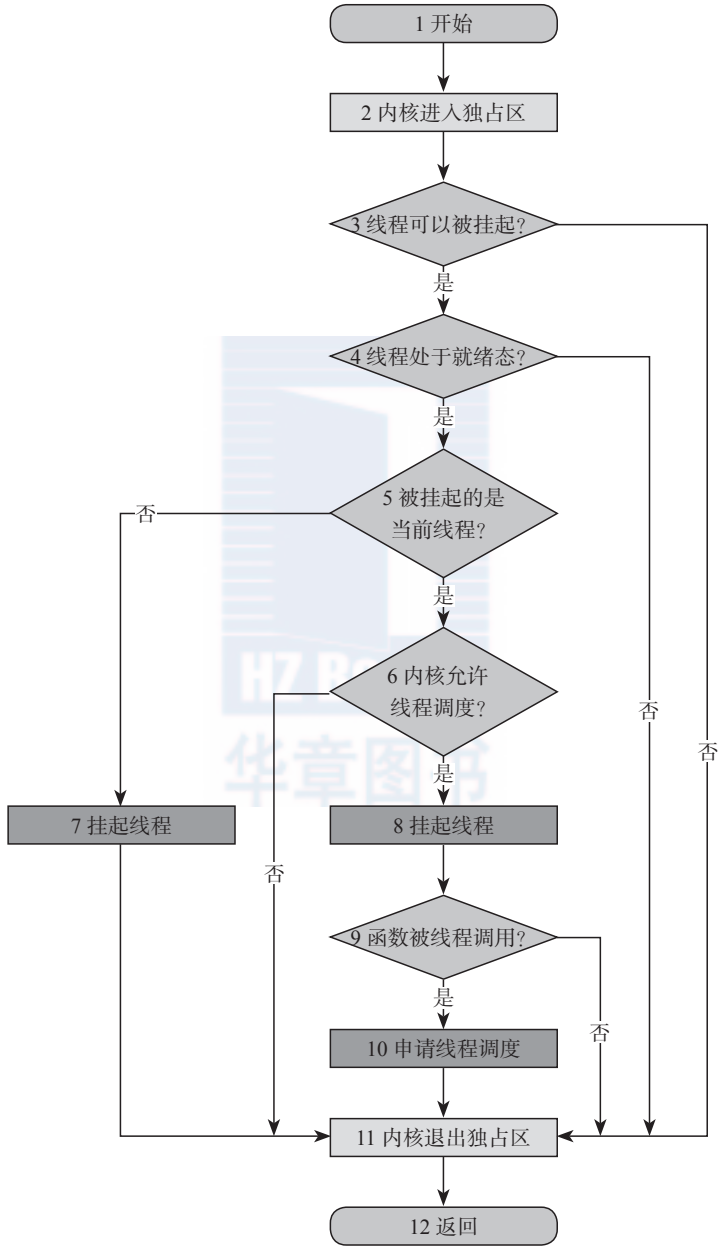

图 2-8　线程挂起流程图

线程挂起伪代码如代码清单 2-7 所示。

代码清单2-7：线程挂起伪代码

```
1.  线程挂起
2.  {
3.      关闭中断;
4.      如果线程可以被挂起
5.      {
6.          如果线程状态为就绪态
7.          {
8.              如果是当前线程
9.              {
10.                 如果内核此时没有禁止线程调度
11.                 {
12.                     挂起线程
13.                     如果函数被线程调用
14.                     {
15.                         申请线程调度
16.                     }
17.                 }
18.             }
19.             否则
20.             {
21.                 挂起线程
22.             }
23.         }
24.     }
25.     打开中断;
26. }
```

2.4.5 线程解挂

Trochili RTOS 实现了线程解挂的功能。跟线程挂起相比较，这里不需要考虑内核是否允许线程调度，这是因为当前线程的挂起必然导致线程调度。而在线程解挂的流程，并不一定进行线程切换，不需要关心内核是否允许线程调度。在线程挂起的流程中，很早就对内核调度是否使能进行判断，而解挂时，则推迟到最后尝试线程调度阶段才处理。

线程解挂操作的核心步骤是：

- ❏ 把线程从内核辅助队列中移到内核就绪队列
- ❏ 设置线程状态为就绪
- ❏ 必要时，向内核申请线程调度

整个线程解挂的过程需要考虑以下几个问题：

- ❏ 线程是否允许被解挂
- ❏ 线程是否处于挂起状态
- ❏ 函数被线程调用还是被 ISR 调用

线程解挂操作的主要流程如图 2-9 所示。

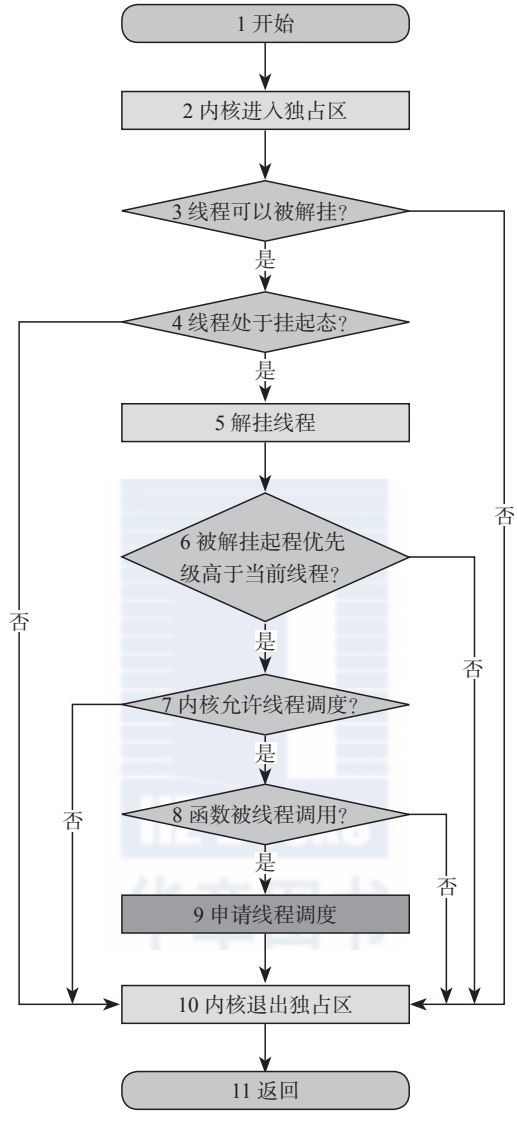

图 2-9　线程解挂流程图

线程解挂流程图分析：

STEP3　如果某个线程没有配置成可解挂，那么对该线程的解挂操作会失败。

STEP4　解挂操作只对处于挂起状态的线程有效。

STEP5　解挂线程，主要是对线程队列和状态的操作。

STEP6　如果被解挂的线程的优先级高于当前线程的优先级则需要检查是否需要调度。

STEP7　如果内核没有关闭线程调度则需要考虑线程调度的问题。

STEP8　如果线程解挂函数是被 ISR 调用则不需要发出线程调度。

STEP9　向内核发出线程调度请求。

STEP10　内核代码退出独占区，在中断退出时，内核可能响应线程调度请求。

线程解挂的伪代码如代码清单 2-8 所示。

代码清单2-8：线程解挂伪代码

```
1.  线程解挂
2.  {
3.      关闭中断;
4.      如果线程可以被解挂
5.      {
6.          如果线程状态为挂起
7.          {
8.              线程离开内核辅助线程队列
9.              如果是当前线程
10.             {
11.                 线程在内核就绪线程队列的位置为队列头
12.             }
13.             将线程加入内核就绪线程队列
14.             设置线程状态为就绪
15.
16.             如果被操作的线程的优先级高于线程就绪队列的最高优先级，
17.             并且内核此时并没有关闭线程调度，
18.             并且本函数被线程调度：
19.             {
20.                 那么就需要申请线程调度
21.             }
22.         }
23.     }
24.     打开中断;
25. }
```


2.4.6　线程延时

在线程运行的过程中，有可能需要延时一段时间，这时需要使用线程延时函数，它启动当前线程的软定时器，将其加入内核定时器队列，随后把线程加入内核线程辅助队列，直到定时器到时，内核自动唤醒该线程，该线程继续运行。

需要注意的是，每个线程都拥有一个软定时器，线程延时和资源时限等待功能都是通过这个软定时器实现的。另外，因为软定时器是通过系统时钟节拍中断实现的，所以它的精度是有限定的，在 10ms 的时钟节拍下，软定时器的精度是上下 10ms。

Trochili RTOS 实现了线程延时的功能。只有处于就绪状态的线程可以被线程和 ISR 延时，如果当前线程被延时，内核会立刻发出线程调度请求；同样，只有处于延时状态的线程才可以被线程和 ISR 取消延时，如果延时取消后的线程的优先级高于当前线程的优先级则内核尝试发起线程调度请求。

线程延时操作的核心步骤是：

❑ 把线程从内核就绪队列中移到内核辅助队列

❑ 设置线程状态为延时
❑ 启动线程定时器
❑ 必要时向内核发出线程调度请求

整个线程延时的过程需要考虑下面几个问题：

❑ 线程是否允许被延时
❑ 线程是否处于就绪态，是否是当前线程被延时
❑ 内核是否允许线程调度
❑ 函数被线程调用还是被 ISR 调用

线程延时操作的主要流程如图 2-10 所示。

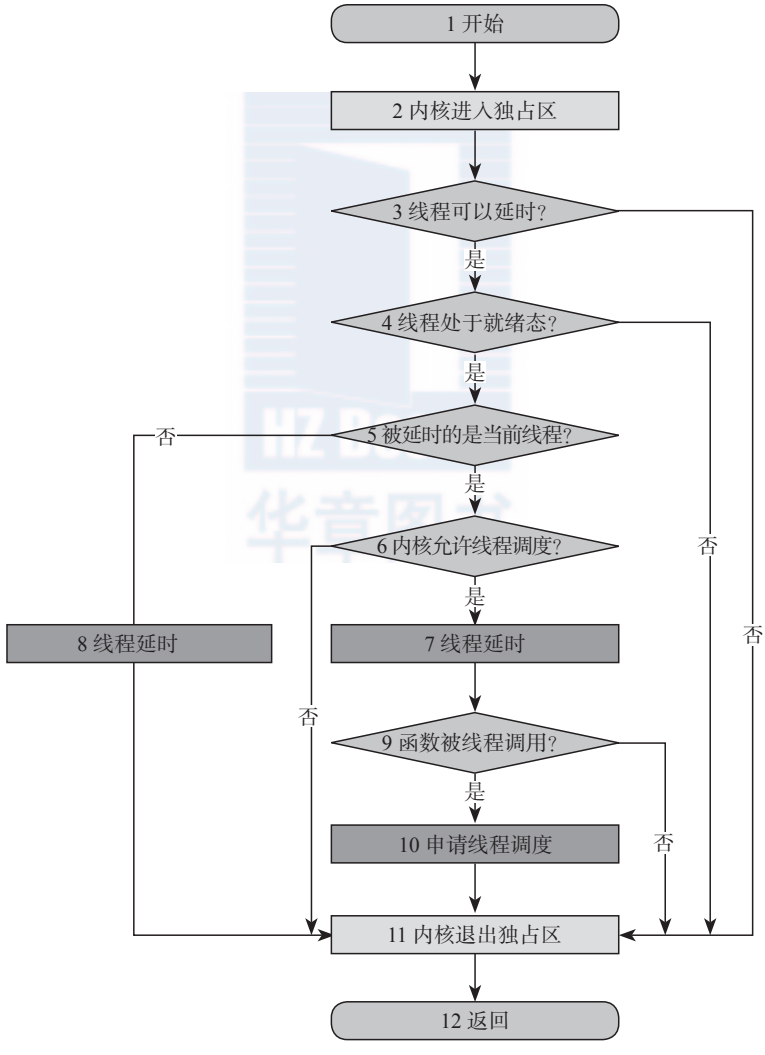

图 2-10　线程延时流程图

线程延时流程图分析：

STEP3　如果某个线程没有配置成可延时，那么对该线程的延时操作会失败。

STEP4　延时操作只对处于就绪状态的线程有效。

STEP5　如果被延时的线程是当前线程则说明延时操作会引起线程调度。

STEP6　但如果此时内核不允许线程调度则本次延时操作失败。

STEP7　当前线程延时，主要是对线程队列和状态的操作；同时启动线程定时器。

STEP9　判断函数是不是被线程调度。

STEP10　如果是则向内核发出线程调度请求。

STEP11　内核代码退出独占区后，可能发生线程调度。

线程延时的功能是和线程定时器紧密相关的，在第 8 章中会重点介绍。线程延时的伪代码如代码清单 2-9 所示。

代码清单2-9：线程延时伪代码

```
1.  线程延时
2.  {
3.      关闭中断;
4.      如果线程可以被延时
5.      {
6.          如果线程状态为就绪态
7.          {
8.              如果是当前线程
9.              {
10.                 如果内核此时没有禁止线程调度
11.                 {
12.                     将线程延时
13.                     如果本函数是被线程调用
14.                     {
15.                         需要申请线程调度
16.                     }
17.                 }
18.             }
19.             否则
20.             {
21.                 将线程延时
22.             }
23.         }
24.     }
25.     打开中断;
26. }
```

2.4.7　线程延时取消

这个功能和线程延时正好相反，要把已经休眠的线程重新调整为就绪态，取消先前的休眠操作。

线程延时取消操作的核心步骤是：

- ❑ 把线程从内核辅助队列中移到内核就绪队列
- ❑ 设置线程状态为就绪
- ❑ 停止线程定时器
- ❑ 必要时向内核申请线程调度

整个线程延时取消的过程需要考虑下面几个问题：

- ❑ 线程是否允许被取消休眠
- ❑ 线程是否处于休眠状态
- ❑ 函数被线程调用还是被 ISR 调用

跟线程延时相比较，这里不需要考虑内核是否允许线程调度，这是因为当前线程的延时必然导致线程调度，而在线程延时取消的流程，并不一定进行线程切换，不需要关心内核是否允许线程调度。在线程延时的流程中，很早就对内核调度是否使能进行判断，而延时取消时，则推迟到最后尝试线程调度阶段才处理。

线程延时取消操作的主要流程如图 2-11 所示。

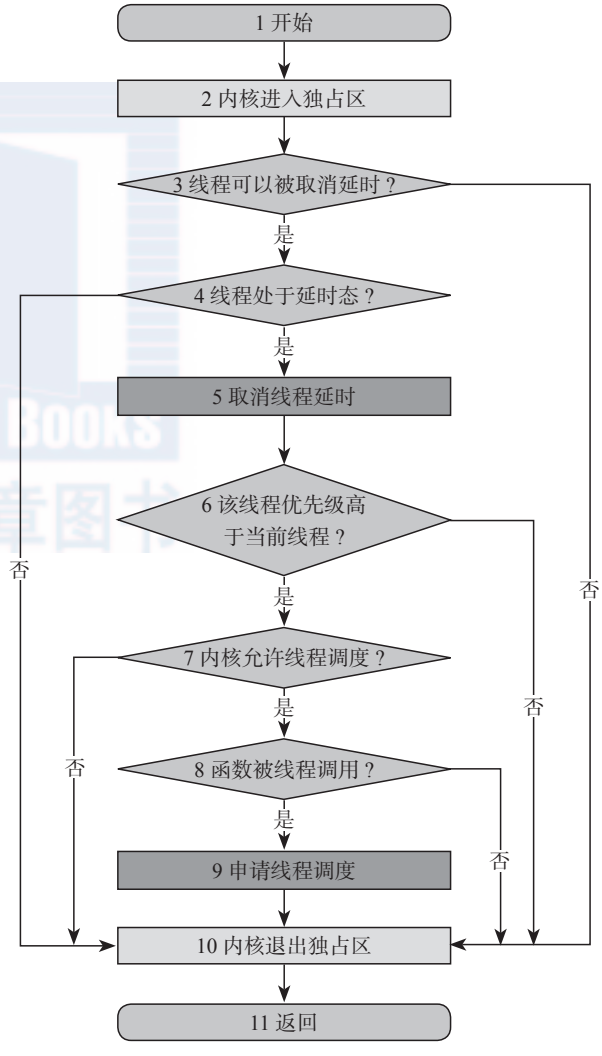

图 2-11 线程延时取消流程图

线程延时取消流程图分析：

STEP3 如果某个线程没有配置成延时可取消，那么对该线程的操作会失败。

STEP4 取消延时只对处于延时状态的线程有效。

STEP5 取消线程延时，主要是对线程队列和状态的操作；同时关闭线程定时器。

STEP6 如果被取消延时的线程的优先级高于当前线程的优先级则需要检查是否需要调度。

STEP7 如果内核没有关闭线程调度，则需要考虑线程调度的问题。

STEP8 如果线程延时取消函数是被 ISR 调用，则不需要发出线程调度。

STEP9　向内核发出线程调度请求。

STEP10　内核代码退出独占区，在中断退出时，内核可能响应线程调度请求。

线程延时取消的伪代码如代码清单 2-10 所示。

代码清单2-10：线程延时取消伪代码

```
1.  线程延时取消
2.  {
3.      关闭中断;
4.      如果线程可以被取消延时
5.      {
6.          如果线程状态为延时
7.          {
8.              线程离开内核辅助线程队列
9.              如果是当前线程
10.             {
11.                 线程将被放入内核就绪线程队列的队列头
12.             }
13.             将线程加入内核就绪线程队列
14.             设置线程状态为就绪
15.
16.             如果被操作的线程的优先级高于线程就绪队列的最高优先级，
17.             并且内核此时并没有关闭线程调度，
18.             并且本函数被线程调度：
19.             {
20.                 需要申请线程调度
21.             }
22.         }
23.     }
24.     打开中断;
25. }
```

2.4.8　线程主动调度

Trochili RTOS 实现了用户级线程主动调度功能，即当前线程通过该 API 尝试将处理器控制权释放，转让给别的线程。当前线程只能把处理器让给同优先级的线程。

IDLE 守护线程是 Trochili RTOS 中具有最低优先级的就绪线程，因为它一直处于就绪状态，所以某个就绪线程如果调用 TclYieldThread () 函数，不会导致内核中没有就绪线程的错误。

用户级线程主动调度的核心步骤是：

- ❑ 查找新的就绪线程用于调度
- ❑ 向内核发出线程调度请求

用户级线程主动调度的过程需要考虑下面几个问题：

- ❑ 当前线程是否允许主动发起调度
- ❑ 内核当前是否允许线程调度

❑ 只能被当前线程调用执行

线程主动调度的操作的主要流程如图 2-12 所示。

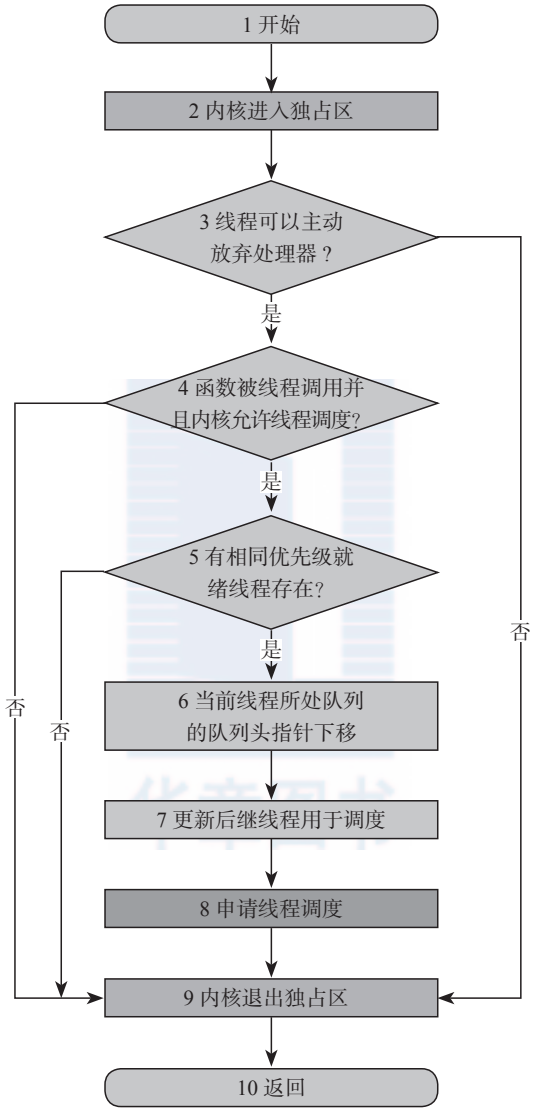

图 2-12　线程主动调度流程图

线程主动调度流程图分析：

STEP3　如果当前线程不支持主动放弃处理器，那么操作会失败。

STEP4　如果函数不是被线程调用或者内核不允许线程调度则退出。

STEP5　判断是否有相同优先级的就绪线程。

STEP6　如果有相同优先级的就绪线程，则将当前线程所处队列的头指针下移。

STEP7　得到新的就绪线程等待调度。

STEP8　向内核发出线程调度请求。

STEP9　内核代码退出独占区后，可能发生中断，发生线程调度。

用户级线程主动调度的伪代码如代码清单 2-11 所示。

代码清单2-11：用户级线程主动调度伪代码

```
1.  用户级线程调度
2.  {
3.      关闭中断;
4.      如果当前线程可以调用本函数
5.      {
6.          如果是线程调用本函数
7.          并且内核允许线程调度
8.          {
9.              如果有和当前线程优先级相同的线程存在
10.             {
11.                 当前线程所处内核就绪线程队列分队列的头指针下移
12.                 重新得出最佳就绪线程
13.                 向内核申请线程调度
14.             }
15.         }
16.     }
17.     打开中断;
18. }
```

2.4.9　线程优先级设定

Trochili RTOS 中，实现了线程动态优先级的功能。用户线程可以在运行过程中主动地修改自己或者其他线程的优先级。因为内核是根据线程的优先级来安排线程所处队列的，所以修改线程优先级的过程需要考虑以下细节问题：

- ❑ 线程所处线程队列和状态
- ❑ 线程是否为当前线程，当前线程是否处于就绪队列
- ❑ 线程优先级变化会导致它会移动到哪些分队列
- ❑ 什么情况下会引起线程调度

为了方便，我们首先假设内核中就绪线程分布情况如图 2-13 所示。

注意为了演示需要，刻意在每个队列尾加入了一个空的线程节点，这样可以更形象地表示出线程的队列和位置迁移。但在实际系统内部，不会有这样的空节点存在；另外在内核的线程就绪队列和辅助队列里，每个分队列的节点都是组织成双向循环链表的。在这里为了图像的清晰，也省略了队列头和队列尾节点之间的双向连线。

在图 2-13 的演示中，线程 Thread31 为当前线程，Thread32、Thread33 和它处于同一个就绪线程分队列中。

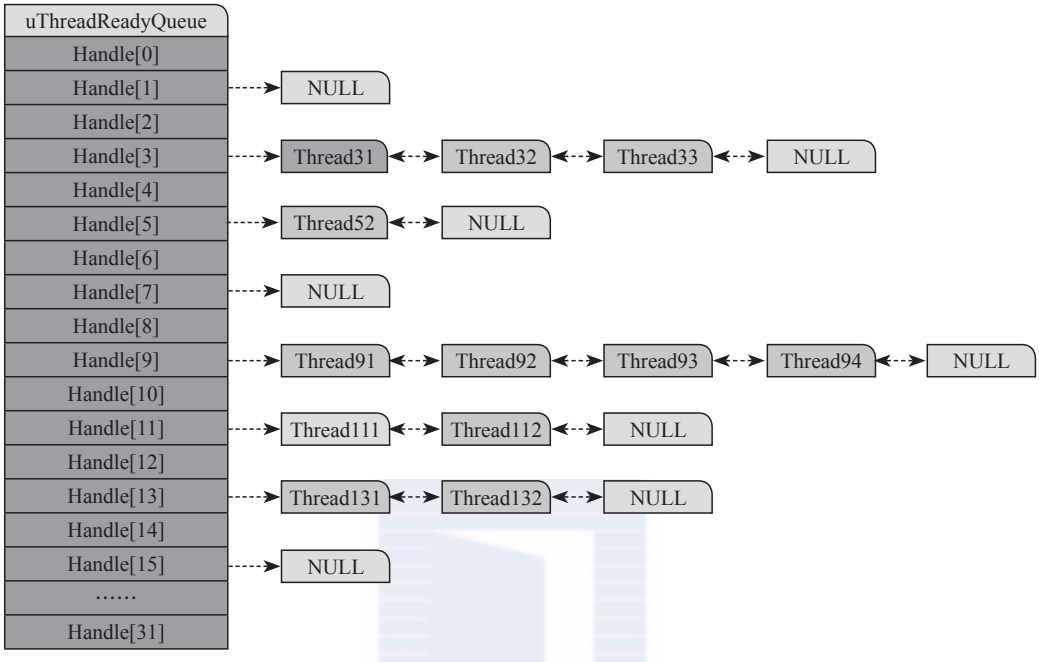

图 2-13　修改线程优先级演示 1

我们以图 2-13 为基准，来分析一下内核就绪线程队列中的线程的优先级的变化情况。首先来分析一下当前线程之外的其他最高就绪优先级的线程（即图 2-13 中的 Thread32 和 Thread33 这样的线程）的情况，如果它们的优先级被修改，那么可能发生的移动情况如图 2-14 所示。

图 2-14 说明了最高就绪优先级的非队列头的线程的迁移的可能情况：

❏ 优先级提升到更高的级别，成为队列头（孤节点），抢占当前线程；

❏ 优先级降低到低级别，成为队列头（孤节点），不会引起线程调度；

❏ 优先级降低到低级别，成为队列尾，不会引起线程调度。

Thread32 节点向 NULL1 节点的迁移过程说明 Thread32 的优先级提高到 1，因为优先级为 1 的队列为空（必须是空的），所以成为该队列的头节点，并且需要线程调度。

Thread32 节点向 NULL7 节点的迁移过程说明 Thread32 的优先级降低到 7，因为优先级为 7 的队列为空，所以成为该队列的头节点，不需要线程调度。

Thread32 节点向 NULL5 节点的迁移过程说明 Thread32 的优先级降低到 5，因为优先级为 5 的队列已经有线程存在，所以成为该队列的尾节点，不需要线程调度。

Thread33 和 Thread32 唯一不同的地方就是它们一个处于队列尾，一个处于队列中。当优先级变化，引起其所属分队列的变化的情况是一样的。

再来分析一下非最高就绪优先级的线程的情况，因为每个线程的迁移路线都比较多，所以分开演示某个队列中不同位置的节点的迁移情况。

如果线程处于队首，则其队列迁移如图 2-15 所示。

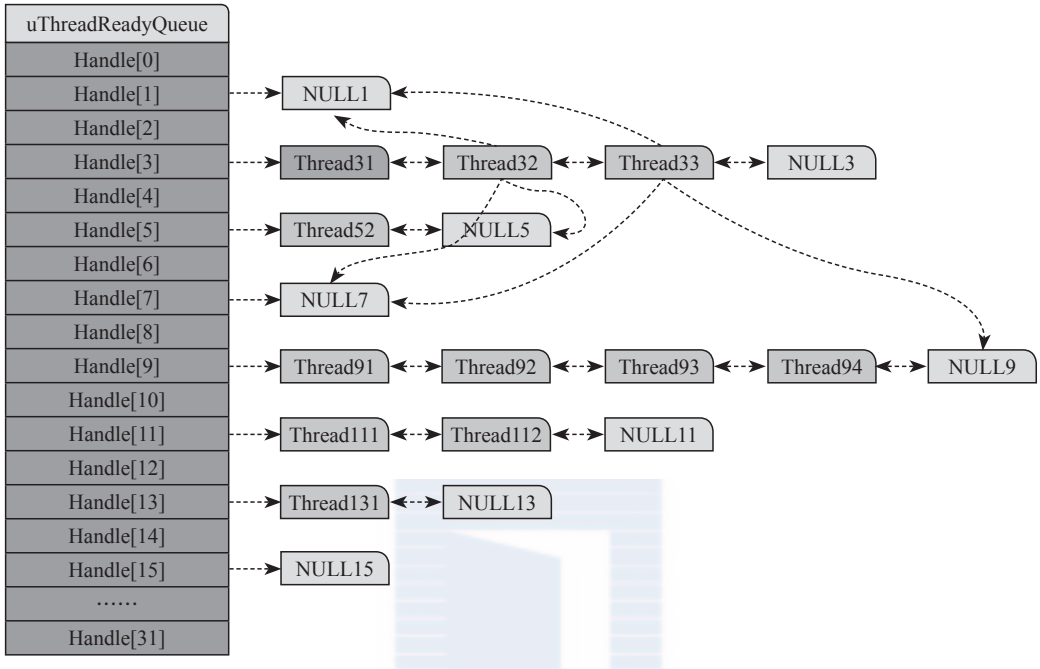

图 2-14　修改线程优先级演示 2

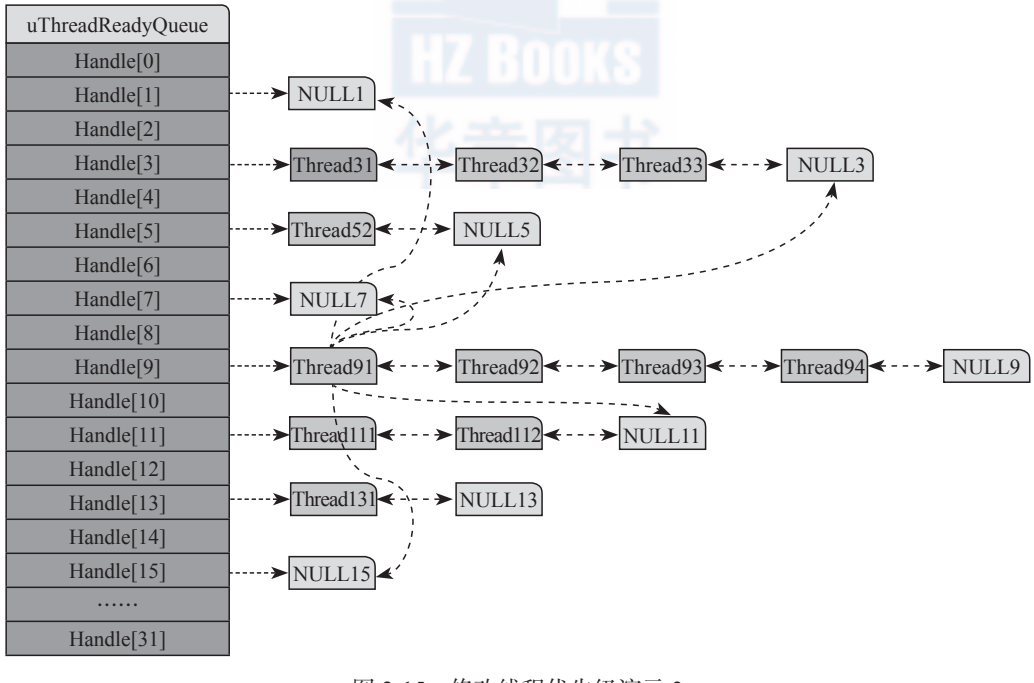

图 2-15　修改线程优先级演示 3

非最高就绪优先级的线程，如果处于队中，则其队列迁移如图 2-16 所示。

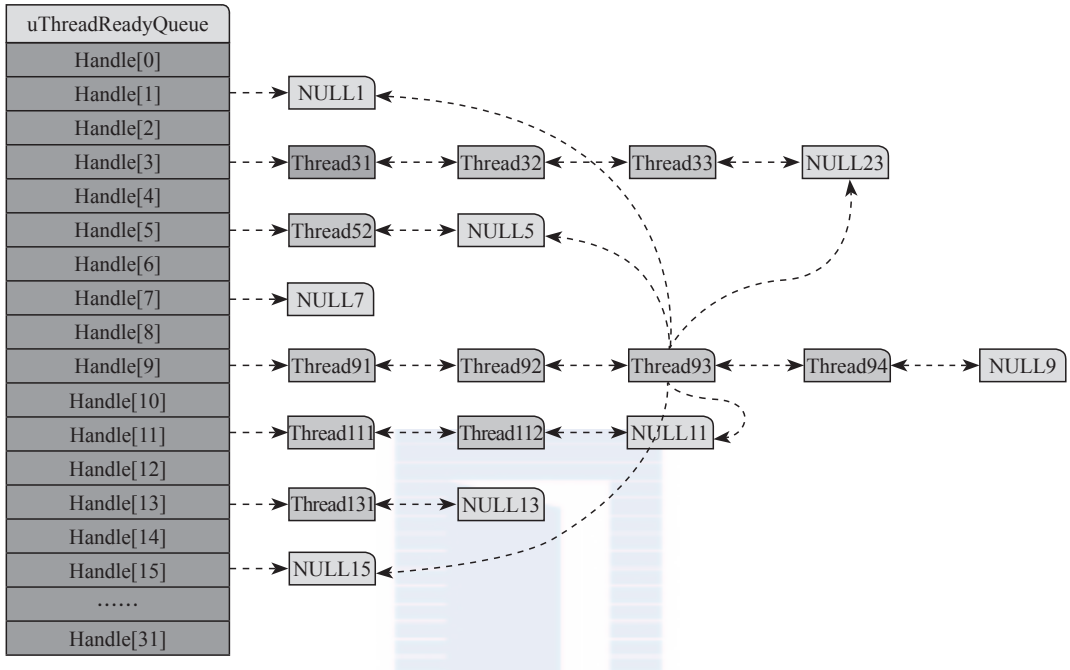

图 2-16　修改线程优先级演示 4

非最高就绪有限级的就绪线程，如果处于队尾，则其队列迁移如图 2-17 所示。

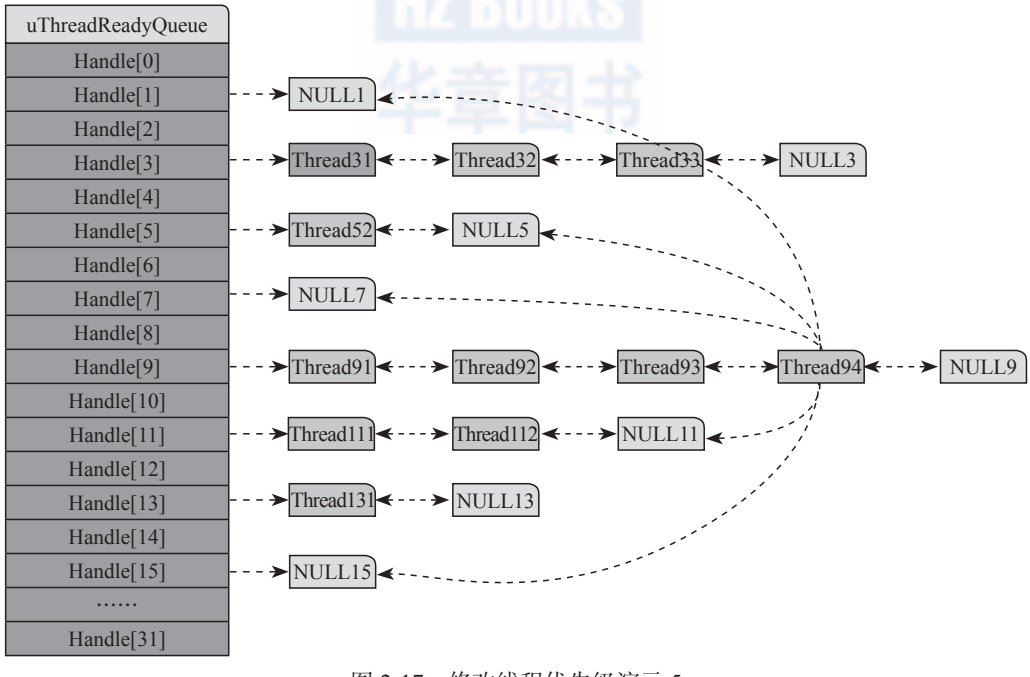

图 2-17　修改线程优先级演示 5

图 2-15 至图 2-17 说明了非就绪优先级的线程的迁移的可能情况：

- ❑ 优先级提升到比当前最高就绪优先级更高的级别，抢占当前线程；
- ❑ 优先级提升到当前最高就绪优先级，成为队列尾，不会引起线程调度；
- ❑ 优先级提升，但低于最高的级别，成为队列头（孤节点），不会引起线程调度；
- ❑ 优先级提升，但低于最高的级别，成为队列尾，不会引起线程调度；
- ❑ 优先级降低到低级别，成为队列头（孤节点），不会引起线程调度；
- ❑ 优先级降低到低级别，成为队列尾，不会引起线程调度。

Thread91 节点向 NULL1 节点的迁移过程说明 Thread91 的优先级提高到 1，因为优先级为 1 的队列为空（必须是空的），所以成为该队列的头节点，并且需要线程调度。

Thread91 节点向 NULL3 节点的迁移过程说明 Thread91 的优先级提高到 3，成为当前最高就绪优先级，因为优先级为 3 的线程队列不为空，所以成为该队列的尾节点，不需要线程调度。

Thread91 节点向 NULL5 节点的迁移过程说明 Thread91 的优先级上升到 5，因为优先级为 5 的队列不为空，所以成为该队列的尾节点，不需要线程调度。

Thread91 节点向 NULL7 节点的迁移过程说明 Thread91 的优先级上升到 7，因为优先级为 7 的队列为空，所以成为该队列的头节点，不需要线程调度。

Thread91 节点向 NULL11 节点的迁移过程说明 Thread91 的优先级降低到 11，因为优先级为 11 的队列已经有线程存在，所以成为该队列的尾节点，不需要线程调度。

Thread91 节点向 NULL15 节点的迁移过程说明 Thread91 的优先级降低到 15，因为优先级为 15 的队列没有线程存在，所以成为该队列的尾节点，不需要线程调度。

Thread91、Thread93 和 Thread94 唯一不同的地方就是它们在队列中的位置。但当优先级变化，引起其所属分队列的变化的情况是一样的。

现在我们再来看一下，当前线程的优先级变化会导致怎么样的线程分队列迁移。因为当前线程并不一定一直处于就绪队列，所以必须说明这里讲的是当前线程处于就绪队列的情况。当前线程和就绪线程相比较，不同的地方在于当前线程在分队列中迁移时，是插入队列头的，即无论它处于哪个队列中，都保持它的队列头位置不变。而其他的就绪线程则是追加到队列尾的。

我们首先按照图 2-18 来演示当前线程的情况。

当前线程 Thread31 的优先级变化有以下几种情况，因为当前线程要放在队列头，所以它的迁移情况比较简单，如图 2-19 所示。

Thread31 节点向 NULL1 节点的迁移过程说明 Thread31 的优先级提高到 1，Thread31 成为该队列的头节点，并且不需要线程调度。

Thread31 节点向 Thread51 节点前的迁移过程说明 Thread31 的优先级降低到 5，Thread31 成为该队列的头节点，并且需要线程调度。

Thread31 节点向 NULL7 节点的迁移过程说明 Thread31 的优先级降低到 7，Thread31 成为该队列的头节点，并且需要线程调度。

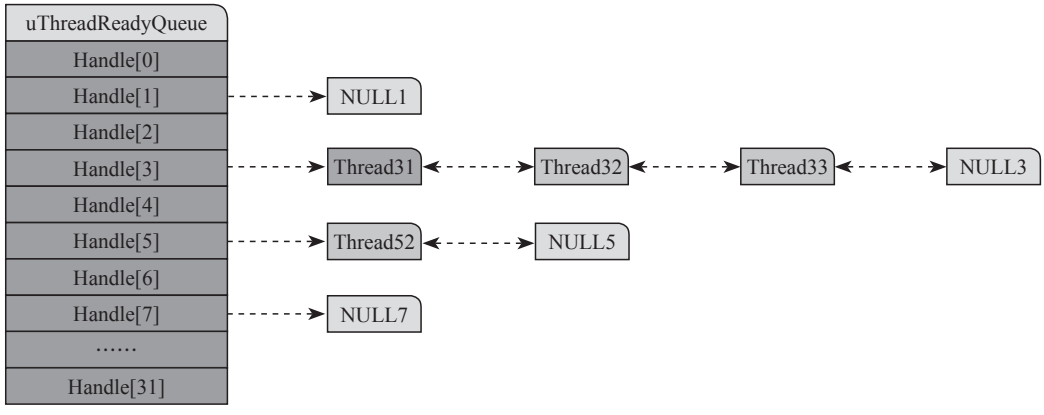

图 2-18　修改当前线程优先级演示 6

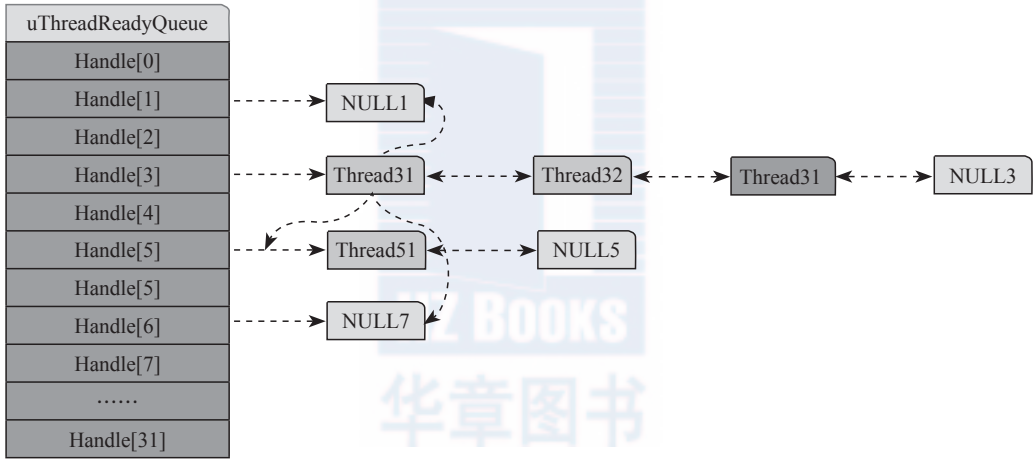

图 2-19　当前线程处于队列头时修改优先级

另外，程序在执行用户级线程调度时，会把当前线程调整到就绪线程队列的队尾；此外在时间节拍中断处理过程中，有时也会把当前线程调整到就绪线程队列的队尾。在这两种情况下，如果调整当前线程的优先级，则当前线程的位置的变化如图 2-20 所示。

以上讲的都是被修改优先级的线程处于内核线程就绪队列的情况。处于阻塞状态的线程的优先级变化会在第 3 章详细介绍。而除了就绪和阻塞之外的其他状态的线程都处于内核线程辅助队列中，没有因优先级引起的调度抢占关系，优先级变化也没有引起线程调度可能。所以可以直接修改线程的优先级。

就绪设定线程优先级的整体流程如下：

❏ 检查线程是否允许修改优先级

❏ 判别线程状态，区分处理处于不同状态的线程的优先级

❏ 如果是当前线程的优先级被修改，则需要进行调度处理

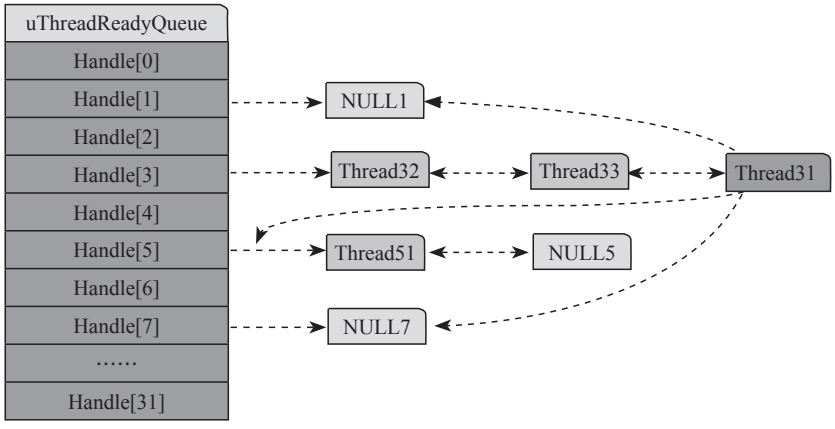

图 2-20 当前线程处于队尾时修改优先级

流程图分析：

STEP2 如果某个线程没有配置成优先级可设置，那么对该线程的操作会失败。

STEP4 处于阻塞状态的线程有单独的函数来处理。

STEP5 处于就绪状态的线程也有单独的流程来处理。

STEP6 处于其他状态的线程，可直接修改线程优先级。

STEP9 在就绪线程的处理过程中，检查函数是否被线程调度并且检查内核是否允许线程调度。

STEP10 如果是被线程调度的，那么计算最高就绪优先级。

STEP11 如果最高就绪优先级比当前线程的优先级还要高，那么需要进行线程调度。

STEP12 向内核申请线程调度。

STEP13 内核代码退出独占区后可能发生中断。在中断退出时可能发生线程调度。

代码清单2-12：线程优先级修改伪代码

```
1.  修改线程优先级
2.  {
3.      关闭中断;
4.      如果线程可以被修改优先级
5.      {
6.          针对不同状态的线程修改其优先级
7.          继续处理状态为就绪态的线程
8.          {
9.              如果是线程调用本函数 并且 内核没有关闭线程调度
10.             {
11.                 得到当前就绪队列的最高就绪优先级
12.                 如果有更高就绪优先级
13.                 {
14.                     向内核申请线程调度
15.                 }
16.             }
17.     }
18.     打开中断;
19. }
```

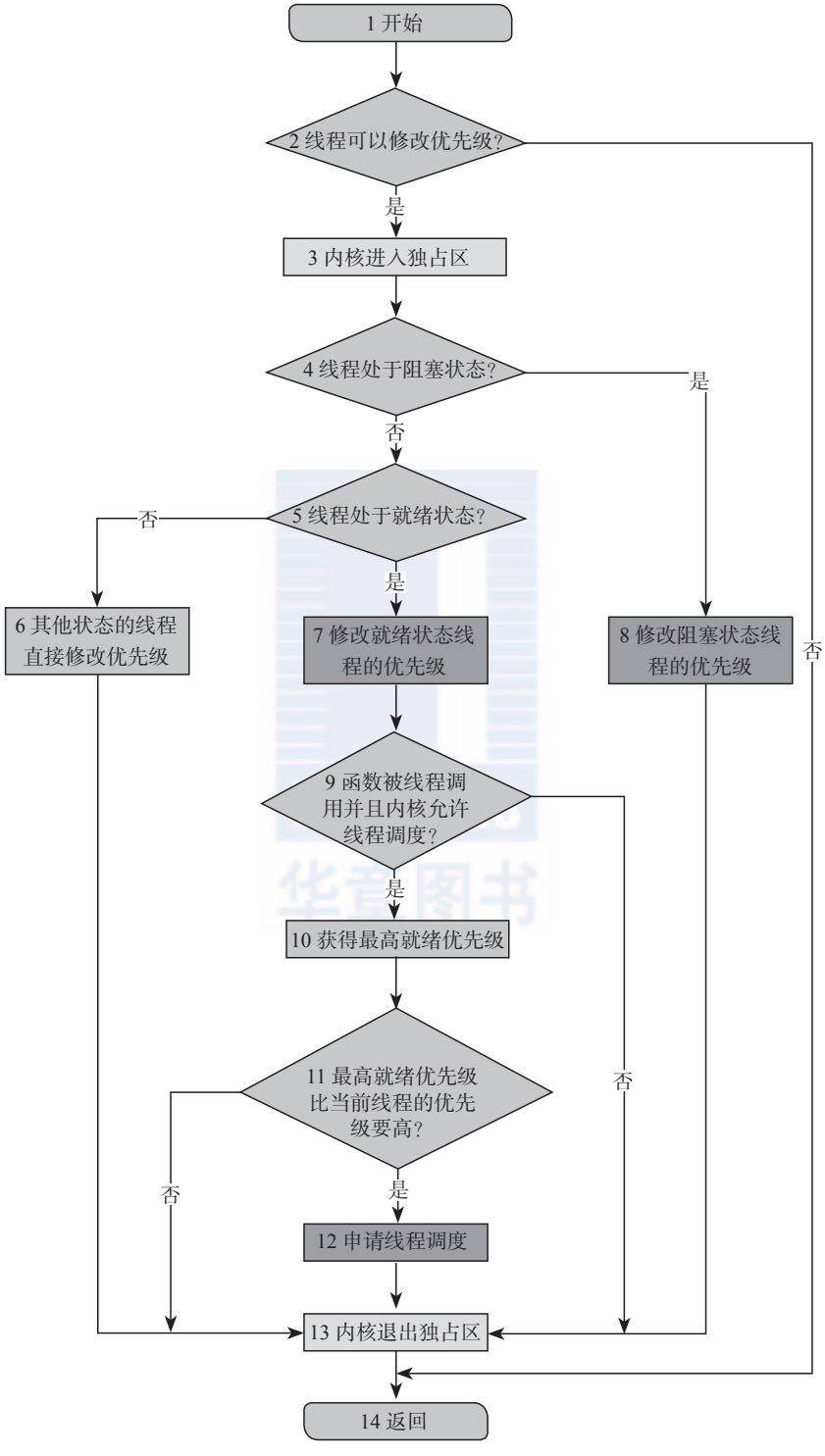

图 2-21 修改线程优先级

2.4.10 线程时间片修改

在同优先级的线程间，时间片长度决定了线程持有处理器的时间，各线程按照时间片比例轮流占用处理器。Trochili RTOS 实现了线程时间片修改的功能，在内核运行时，能够动态调用线程的时间片。

在修改线程时间片时，需要检查线程是否允许修改时间片，如果可以则直接修改。并保持线程在它当前的线程队列中，同时保持状态不变。

修改线程时间片长度的流程如图 2-22 所示。

流程图分析：

STEP3 如果某个线程没有配置成时间片可变，那么对该线程的时间片调整操作会失败。

STEP4 时间片调整。

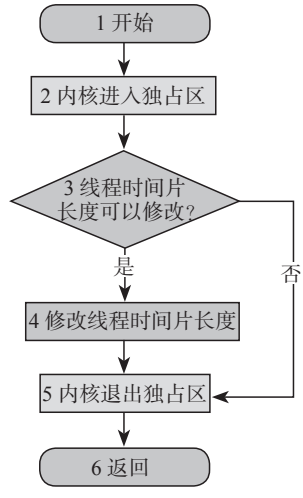

图 2-22 修改线程时间片长度

代码清单2-13：修改线程时间片伪代码

```
1.  修改线程时间片
2.  {
3.      关闭中断
4.      如果线程时间片可以被修改
5.      {
6.          如果新的时间片大于原有时间片
7.          {
8.              修正本轮时间片剩余计数（算法 1）
9.          }
10.         否则
11.         {
12.             修正本轮时间片剩余计数（算法 2）
13.         }
14.     }
15.     打开中断;
16. }
```

2.5 系统守护线程

Trochili RTOS 提供了两个系统守护线程，IDLE 线程和 Timer 线程。其中 IDLE 线程是必不可少的。当系统中没有用户线程就绪时，将由它来占用处理器；而 Timer 线程则可以根据用户需求来剪裁配置，如果用户不需要定时操作那么可以通过内核配置文件取消用户定时器功能。

IDLE 线程同时是内核多任务机制的启动线程。因为 IDLE 线程的优先级被设置为 31，

即最低优先级，所以最后 IDLE 线程总会被其他线程抢占。启动多任务之后，IDLE 线程的行为不确定，由用户注册的 HOOK 函数来定义，一般来说可以做低功耗等操作。

Timer 守护线程用来处理用户回调类型定时器的，即当用户定时器计数结束时，Timer 守护线程会调用用户定时器注册的回调函数，代理定时器任务。Timer 守护线程的优先级默认为 1。

2.6 线程应用演示

本节我们通过在评估板上的实例来演示有关线程调度和管理的 API。因为每个例子中都包括了线程的创建，所以这里不再单独介绍如何创建线程。示例程序尽量设计得简单，主要依靠评估板上的 LED 灯来演示代码的执行路径。

2.6.1 线程激活和休眠演示

上面介绍了线程激活和休眠的功能设计和 API 原型。下面我们就演示一下这两种功能如何使用。我们要实现的功能是通过几个线程的合作，在 Colibri 评估板上按照 1 秒的时间间隔打开和关闭 LED 设备。

在下面的例子中，共有三个线程，线程 ThreadLEDOn 用来保持 LED 设备开启，线程 ThreadLEDOff 用来保持 LED 设备关闭，线程 ThreadCTRL 则用来控制另外两个线程的运行。在用户初始化函数中，对这三个线程都进行了初始化。但是只激活了 ThreadCTRL。所以当内核初始化结束后，ThreadCTRL 首先运行。ThreadCTRL 线程按照 1 秒的时间间隔进行延时，在延时前后则控制另外两个线程的休眠与激活。因为 ThreadCTRL 的优先级高，所以该线程从延时中返回后立刻通过优先级抢占获得处理器控制权，从而可以及时地控制另外两个线程的运行。具体代码如代码清单 2-14 所示。

代码清单2-14：线程激活和休眠

```
1.  #include "trochili.h"
2.  #include "example.h"
3.
4.  #if (EVB_EXAMPLE == CH2_THREAD_EXAMPLE2)
5.
6.  /* LED设备参数 */
7.  #define LED_ON     (1)
8.  #define LED_OFF    (0)
9.  #define LED_INDEX (2)
10.
11. /* 用户线程参数 */
12. #define THREAD_LED_STACK_SIZE    (512)
13. #define THREAD_LED_PRIORITY      (5)
14. #define THREAD_LED_SLICE         (20)
15.
```

```
16.  #define THREAD_CTRL_STACK_SIZE   (512)
17.  #define THREAD_CTRL_PRIORITY     (4)
18.  #define THREAD_CTRL_SLICE        (20)
19.
20.  /* 用户线程定义 */
21.  static TThread ThreadLEDOn;
22.  static TThread ThreadLEDOff;
23.  static TThread ThreadCTRL;
24.
25.  /* 用户线程堆栈 */
26.  static TWord ThreadLEDOnStack[THREAD_LED_STACK_SIZE];
27.  static TWord ThreadLEDOffStack[THREAD_LED_STACK_SIZE];
28.  static TWord ThreadCTRLStack[THREAD_CTRL_STACK_SIZE];
29.
30.  /* 线程 LEDOn 的线程函数 */
31.  static void ThreadLEDOnEntry(void* pArg)
32.  {
33.      while (eTrue)
34.      {
35.          EVB_LEDControl(LED_INDEX, LED_ON);
36.      }
37.  }
38.
39.  /* 线程 LEDOff 的线程函数 */
40.  static void ThreadLEDOffEntry(void* pArg)
41.  {
42.      while (eTrue)
43.      {
44.          EVB_LEDControl(LED_INDEX, LED_OFF);
45.      }
46.  }
47.
48.  /* 线程 CTRL 的线程函数 */
49.  static void ThreadCTRLEntry(void* pArg)
50.  {
51.      while (eTrue)
52.      {
53.          /* 激活 LED 设备点亮线程 */
54.          TclActivateThread(&ThreadLEDOn);
55.
56.          /* 控制线程延时 1 秒 */
57.          TclDelayThread(&ThreadCTRL, SEC2TICKS(1));
58.
59.          /* 休眠 LED 设备点亮线程 */
60.          TclDeActivateThread(&ThreadLEDOn);
61.
62.          /* 激活 LED 设备熄灭线程 */
63.          TclActivateThread(&ThreadLEDOff);
64.
65.          /* 控制线程延时 1 秒 */
```

```
66.          TclDelayThread(&ThreadCTRL, SEC2TICKS(1));
67.
68.          /* 休眠 LED 设备熄灭线程 */
69.          TclDeActivateThread(&ThreadLEDOff);
70.      }
71.  }
72.
73.  /* 用户应用入口函数 */
74.  static void EVB_LEDAppEntry(void)
75.  {
76.      /* 配置使能评估板上的 LED 设备 */
77.      EVB_LEDConfig();
78.
79.      /* 初始化 LED 设备点亮线程 */
80.      TclInitThread(&ThreadLEDOn, &ThreadLEDOnEntry, (void*)NULL,
81.                    ThreadLEDOnStack, THREAD_LED_STACK_SIZE,
82.                    THREAD_LED_PRIORITY, THREAD_LED_SLICE);
83.
84.      /* 初始化 LED 设备熄灭线程 */
85.      TclInitThread(&ThreadLEDOff, &ThreadLEDOffEntry, (void*)NULL,
86.                    ThreadLEDOffStack, THREAD_LED_STACK_SIZE,
87.                    THREAD_LED_PRIORITY, THREAD_LED_SLICE);
88.
89.      /* 初始化 CTRL 线程 */
90.      TclInitThread(&ThreadCTRL, &ThreadCTRLEntry, (void*)NULL,
91.                    ThreadCTRLStack, THREAD_CTRL_STACK_SIZE,
92.                    THREAD_CTRL_PRIORITY, THREAD_CTRL_SLICE);
93.
94.      /* 激活 CTRL 线程 */
95.      TclActivateThread(&ThreadCTRL);
96.  }
97.
98.  /* 处理器 BOOT 之后会调用 main 函数 */
99.  int main(void)
100. {
101.     /* 注册处理器初始化函数到内核 */
102.     TclSetCpuEntry(&TCL_STM32F10xInit);
103.
104.     /* 注册板级初始化函数到内核 */
105.     TclSetBoardEntry(&EVB_SetupBoard);
106.
107.     /* 注册板级调试打印函数到内核 */
108.     TclSetTraceRoutine(&EVB_Uart1WriteByte);
109.
110.     /* 注册用户初始化函数到内核 */
111.     TclSetUserEntry(&EVB_LEDAppEntry);
112.
113.     /* 启动内核 */
114.     TclStartKernel();
115.
```

```
116.      return 1;
117. }
118.
119. #endif
```

在 Colibri 评估板上运行上面的程序，可以看到板上的 LED 灯按照 1 秒的时间间隔，依次点亮和熄灭。3 个线程的运行时序如图 2-23 所示。

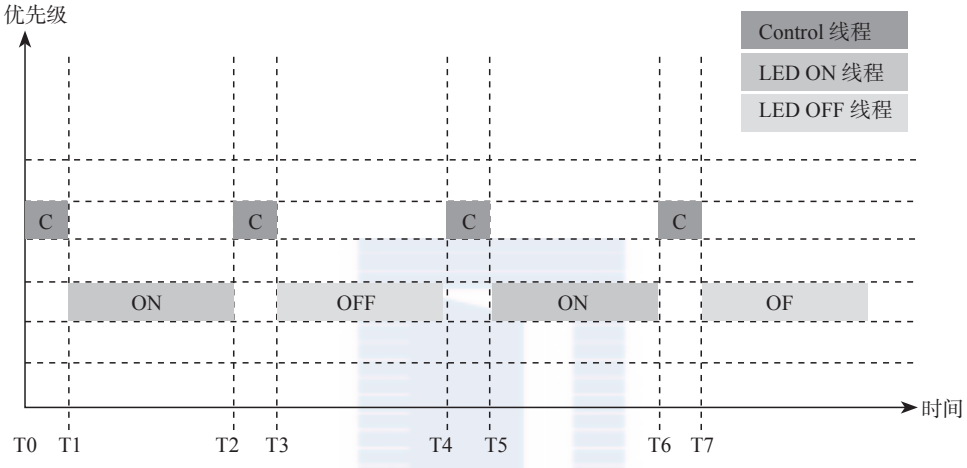

图 2-23 三个线程的运行时序

2.6.2 线程挂起和解挂演示

下面的代码演示了内核中线程挂起和解挂功能，程序首先定义了 2 个应用线程，一个线程用来点亮 LED，一个线程用来熄灭 LED。然后定义了第三个线程作为主控线程，主控线程的功能是按照 1s 的间隔和固定的顺序，分别挂起和解挂前面的 2 个 LED 控制线程，实现 LED 灯按照 1s 的间隔（不是很精准）点亮和熄灭。程序代码如代码清单 2-15 所示。

代码清单2-15：线程挂起和解挂

```
1.  #include "example.h"
2.  #include "trochili.h"
3.
4.  #if (EVB_EXAMPLE == CH2_THREAD_EXAMPLE3)
5.
6.  /* LED 设备参数 */
7.  #define LED_ON     (1)
8.  #define LED_OFF    (0)
9.  #define LED_INDEX  (2)
10.
11. /* 创建线程时需要的参数 */
12. #define THREAD_LED_STACK_SIZE   (512)
13. #define THREAD_LED_PRIORITY     (5)
14. #define THREAD_LED_SLICE        (20)
```

```
15. #define THREAD_CTRL_STACK_SIZE (512)
16. #define THREAD_CTRL_PRIORITY   (4)
17. #define THREAD_CTRL_SLICE      (20)
18.
19. /* 用户线程定义 */
20. static TThread ThreadLEDOn;
21. static TThread ThreadLEDOff;
22. static TThread ThreadCTRL;
23.
24. /* 用户线程栈定义 */
25. static TWord ThreadLEDOnStack[THREAD_LED_STACK_SIZE];
26. static TWord ThreadLEDOffStack[THREAD_LED_STACK_SIZE];
27. static TWord ThreadCTRLStack[THREAD_CTRL_STACK_SIZE];
28.
29. /* 点亮 LED 的线程的主函数 */
30. static void ThreadLEDOnEntry(void* pArg)
31. {
32.     while(eTrue)
33.     {
34.         EVB_LEDControl(LED_INDEX, LED_ON);
35.     }
36. }
37.
38. /* 熄灭 LED 的线程的主函数 */
39. static void ThreadLEDOffEntry(void* pArg)
40. {
41.     while (eTrue)
42.     {
43.         EVB_LEDControl(LED_INDEX, LED_OFF);
44.     }
45. }
46.
47. /* 主控线程的主函数 */
48. static void ThreadCTRLEntry(void* pArg)
49. {
50.     /* 挂起两个 LED 设备控制线程 */
51.     TclSuspendThread(&ThreadLEDOn);
52.     TclSuspendThread(&ThreadLEDOff);
53.
54.     while (eTrue)
55.     {
56.         /* 唤醒 LED 点亮线程 */
57.         TclResumeThread(&ThreadLEDOn);
58.
59.         /* 控制线程休眠 1 秒 */
60.         TclDelayThread(&ThreadCTRL, SEC2TICKS(1));
61.
62.         /* 挂起 LED 点亮线程 */
63.         TclSuspendThread(&ThreadLEDOn);
64.
```

```
65.         /* 唤醒 LED 熄灭线程 */
66.         TclResumeThread(&ThreadLEDOff);
67.
68.         /* 控制线程休眠 1 秒 */
69.         TclDelayThread(&ThreadCTRL, SEC2TICKS(1));
70.
71.         /* 挂起 LED 熄灭线程 */
72.         TclSuspendThread(&ThreadLEDOff);
73.     }
74. }
75.
76. /* 用户应用入口函数 */
77. static void EVB_LEDAppEntry(void)
78. {
79.     /* 配置使能评估板上的 LED 设备 */
80.     EVB_LEDConfig();
81.
82.     /* 初始化 LED 设备控制线程 */
83.     TclInitThread(&ThreadLEDOn, &ThreadLEDOnEntry, (void*)NULL,
84.                   ThreadLEDOnStack, THREAD_LED_STACK_SIZE,
85.                   THREAD_LED_PRIORITY, THREAD_LED_SLICE);
86.
87.     /* 初始化 LED 设备控制线程 */
88.     TclInitThread(&ThreadLEDOff, &ThreadLEDOffEntry, (void*)NULL,
89.                   ThreadLEDOffStack, THREAD_LED_STACK_SIZE,
90.                   THREAD_LED_PRIORITY, THREAD_LED_SLICE);
91.
92.     /* 初始化 CTRL 线程 */
93.     TclInitThread(&ThreadCTRL, &ThreadCTRLEntry, (void*)NULL,
94.                   ThreadCTRLStack, THREAD_CTRL_STACK_SIZE,
95.                   THREAD_CTRL_PRIORITY, THREAD_CTRL_SLICE);
96.
97.     /* 激活 LED 点亮线程 */
98.     TclActivateThread(&ThreadLEDOn);
99.
100.     /* 激活 LED 熄灭线程 */
101.     TclActivateThread(&ThreadLEDOff);
102.
103.     /* 激活主控线程 */
104.     TclActivateThread(&ThreadCTRL);
105. }
106.
107. /* 处理器 BOOT 之后会调用 main 函数，必须提供 */
108. int main(void)
109. {
110.     /* 注册处理器初始化函数到内核 */
111.     TclSetCpuEntry(&TCL_STM32F10xInit);
112.
113.     /* 注册板级初始化函数到内核 */
114.     TclSetBoardEntry(&EVB_SetupBoard);
```

```
115.
116.     /* 注册板级调试打印函数到内核 */
117.     TclSetTraceRoutine(&EVB_Uart1WriteByte);
118.
119.     /* 注册用户初始化函数到内核 */
120.     TclSetUserEntry(&EVB_LEDAppEntry);
121.
122.     /* 启动内核 */
123.     TclStartKernel();
124.
125.     return 1;
126. }
127. #endif
```

上面的程序的执行如图 2-24 所示。

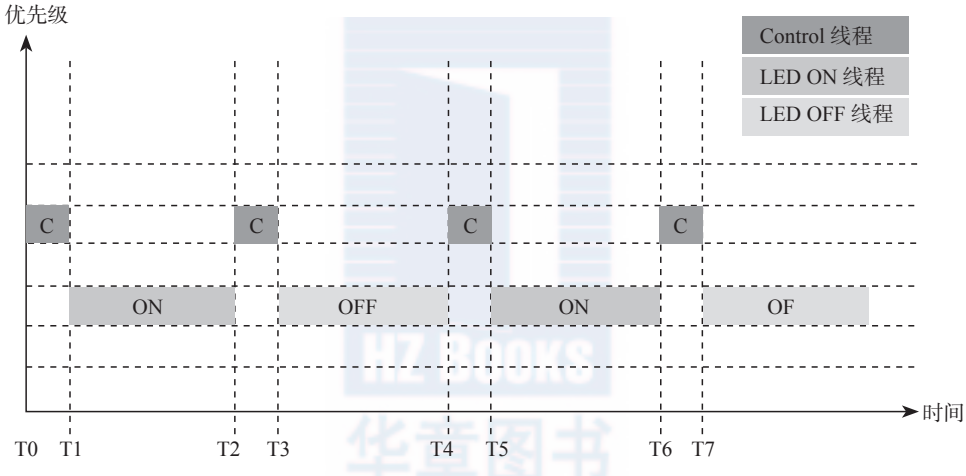

图 2-24 三个线程的运行时序

注意 因为控制线程也占用了运行时间，所以其他 2 个线程并不是完全准确地按照 1s 来点亮和熄灭 LED。

2.6.3 线程延时演示

我们以 Colibri 评估板为例，下面的代码演示了如何使用线程延时 API。我们的目标是在评估板上通过一个 LED 线程不停地实现 1 秒的延时，延时结束后点亮或者熄灭 LED，使得 LED 灯呈现点亮 – 熄灭的闪烁效果。具体代码如代码清单 2-16 所示。

代码清单2-16：线程延时

```
1.  #include "example.h"
2.  #include "trochili.h"
```

```
3.
4.  #if (EVB_EXAMPLE == CH2_THREAD_EXAMPLE4)
5.
6.  /* LED 设备参数 */
7.  #define LED_ON    (1)
8.  #define LED_OFF   (0)
9.  #define LED_INDEX (1)
10.
11. /* 创建线程时需要的参数 */
12. #define THREAD_LED_STACK_SIZE  (256*2)
13. #define THREAD_LED_PRIORITY    (5)
14. #define THREAD_LED_SLICE       (65535)
15.
16. /* 线程定义 */
17. static TThread ThreadLED;
18. /* 线程栈定义 */
19. static TWord ThreadLEDStack[THREAD_LED_STACK_SIZE];
20.
21. /* LED 线程的线程函数 */
22. static void ThreadLEDEntry(void* pArg)
23. {
24.     while (eTrue)
25.     {
26.         EVB_LEDControl(LED_INDEX, LED_ON);
27.         TclDelayThread(uThreadCurrent,SEC2TICKS(1));
28.         EVB_LEDControl(LED_INDEX, LED_OFF);
29.         TclDelayThread(uThreadCurrent,SEC2TICKS(1));
30.     }
31. }
32.
33. /* 用户应用入口函数 */
34. static void EVB_LEDAppEntry(void)
35. {
36.     /* 配置使能评估板上的 LED 设备 */
37.     EVB_LEDConfig();
38.
39.     /* 初始化 LED 设备控制线程 */
40.     TclInitThread(&ThreadLED, &ThreadLEDEntry, (void*)NULL,
41.                   ThreadLEDStack, THREAD_LED_STACK_SIZE,
42.                   THREAD_LED_PRIORITY, THREAD_LED_SLICE);
43.
44.     /* 激活 LED 设备控制线程 */
45.     TclActivateThread(&ThreadLED);
46. }
47.
48. /* 处理器 BOOT 之后会调用 main 函数，必须提供 */
49. int main(void)
50. {
```

```
51.     /* 注册处理器初始化函数到内核 */
52.     TclSetCpuEntry(&TCL_STM32F10xInit);
53.
54.     /* 注册板级初始化函数到内核 */
55.     TclSetBoardEntry(&EVB_SetupBoard);
56.
57.     /* 注册板级调试打印函数到内核 */
58.     TclSetTraceRoutine(&EVB_Uart1WriteByte);
59.
60.     /* 注册用户初始化函数到内核 */
61.     TclSetUserEntry(&EVB_LEDAppEntry);
62.
63.     /* 启动内核 */
64.     TclStartKernel();
65.
66.     return 1;
67. }
68.
69. #endif
```

程序运行结果如图 2-25 所示。

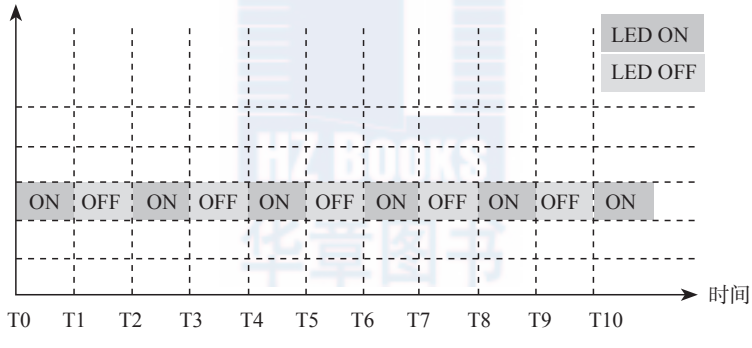

图 2-25　代码清单 2-16 的运行结果

2.6.4　线程主动调度演示

我们以 Colibri 评估板为例，代码清单 2-17 演示了如何使用该 API。我们的目标是在评估板上通过两个独立线程（分别用于点亮和熄灭 LED）的合作，使得某个 LED 灯呈现点亮 – 熄灭的闪烁效果。

我们需要在用户初始化函数里初始化并激活两个线程。在这两个线程函数里，它们首先点亮或者熄灭 LED，然后延时一段时间再调用系统 API 函数 TclYieldThread()，那么每次调用，当前线程都会把处理器控制权交给另一个线程，从而实现两个 LED 控制线程的交互运行。(因为另外两个 LED 控制线程的优先级相同，所以它们在同样的就绪队列里。那么当其中一个线程的时间片运行结束后，它们所处的就绪队列的头指针会下移到另一个

LED 控制线程)。注意这两个 LED 控制线程的时间片都设置到最大，意味着，如果不调用 TclYieldThread() 函数，另一个线程在很长时间内都不会获得运行机会，也就不会出现点亮 – 熄灭的闪灯效果。

代码清单2-17：线程主动发起调度

```
1.  #include "example.h"
2.  #include "trochili.h"
3.
4.  #if (EVB_EXAMPLE == CH2_THREAD_EXAMPLE5)
5.
6.  /* LED 设备参数 */
7.  #define LED_ON    (1)
8.  #define LED_OFF   (0)
9.  #define LED_INDEX (1)
10.
11. /* 创建线程时需要的参数 */
12. #define THREAD_LED_STACK_SIZE  (256*2)
13. #define THREAD_LED_PRIORITY    (5)
14. #define THREAD_LED_SLICE       (0xffffffff)
15.
16. /* 线程定义 */
17. static TThread ThreadLEDOn;
18. static TThread ThreadLEDOff;
19. /* 线程栈定义 */
20. static TWord ThreadLEDOnStack[THREAD_LED_STACK_SIZE];
21. static TWord ThreadLEDOffStack[THREAD_LED_STACK_SIZE];
22.
23. /* 线程做无用操作，起到空转效果 */
24. static Delay(TWord count)
25. {
26.     while(count --);
27. }
28.
29. /* 线程 LEDOn 的线程函数 */
30. static void ThreadLEDOnEntry(void* pArg)
31. {
32.     while (eTrue)
33.     {
34.         EVB_LEDControl(LED_INDEX, LED_ON);
35.         Delay(0xFFFFF);
36.         TclYieldThread();
37.     }
38. }
39.
40. /* 线程 LEDOff 的线程函数 */
41. static void ThreadLEDOffEntry(void* pArg)
42. {
```

```
43.     while (eTrue)
44.     {
45.         EVB_LEDControl(LED_INDEX, LED_OFF);
46.         Delay(0xFFFFF);
47.         TclYieldThread();
48.     }
49. }
50.
51. /* 用户应用入口函数 */
52. static void EVB_LEDAppEntry(void)
53. {
54.     /* 配置使能评估板上的 LED 设备 */
55.     EVB_LEDConfig();
56.
57.     /* 初始化 LED 设备控制线程 */
58.     TclInitThread(&ThreadLEDOn, &ThreadLEDOnEntry, (void*)NULL,
59.                   ThreadLEDOnStack, THREAD_LED_STACK_SIZE,
60.                   THREAD_LED_PRIORITY, THREAD_LED_SLICE);
61.
62.     /* 初始化 LED 设备控制线程 */
63.     TclInitThread(&ThreadLEDOff, &ThreadLEDOffEntry, (void*)NULL,
64.                   ThreadLEDOffStack, THREAD_LED_STACK_SIZE,
65.                   THREAD_LED_PRIORITY, THREAD_LED_SLICE);
66.
67.     /* 激活 LED 设备控制线程 */
68.     TclActivateThread(&ThreadLEDOn);
69.
70.     /* 激活 LED 设备控制线程 */
71.     TclActivateThread(&ThreadLEDOff);
72. }
73.
74. /* 处理器 BOOT 之后会调用 main 函数，必须提供 */
75. int main(void)
76. {
77.     /* 注册处理器初始化函数到内核 */
78.     TclSetCpuEntry(&TCL_STM32F10xInit);
79.
80.     /* 注册板级初始化函数到内核 */
81.     TclSetBoardEntry(&EVB_SetupBoard);
82.
83.     /* 注册板级调试打印函数到内核 */
84.     TclSetTraceRoutine(&EVB_Uart1WriteByte);
85.
86.     /* 注册用户初始化函数到内核 */
87.     TclSetUserEntry(&EVB_LEDAppEntry);
88.
89.     /* 启动内核 */
90.     TclStartKernel();
```

```
91.
92.     return 1;
93. }
94.
95. #endif
```

程序运行过程如图 2-26 所示。

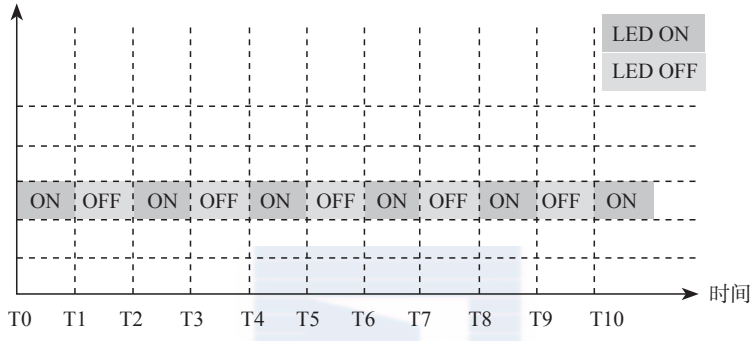

图 2-26 代码清单 2-17 运行结果

2.6.5 线程优先级修改演示

我们以 Colibri 评估板为例，下面的代码演示了如何使用该 API。我们的目标是在评估板上通过两个独立线程的优先级变动，来实现两个 LED 按照先后顺序各自闪烁两次，并循环往复。具体代码如代码清单 2-18 所示。

代码清单2-18：修改线程优先级

```
1.  #include "trochili.h"
2.  #include "example.h"
3.
4.  #if (EVB_EXAMPLE == CH2_THREAD_EXAMPLE6)
5.
6.  /* LED 设备参数 */
7.  #define LED_ON      (1)
8.  #define LED_OFF     (0)
9.  #define LED_INDEX1  (1)
10. #define LED_INDEX2  (2)
11.
12.
13. /* 创建线程时需要的参数 */
14. #define THREAD_LED_STACK_SIZE   (256*2)
15. #define THREAD_LED_PRIORITY     (5)
16. #define THREAD_LED_SLICE        (0xFFFFFFFF)
17.
18. /* 线程定义 */
```

```
19.  static TThread ThreadLED1;
20.  static TThread ThreadLED2;
21.  /* 线程栈定义 */
22.  static TWord ThreadLED1Stack[THREAD_LED_STACK_SIZE];
23.  static TWord ThreadLED2Stack[THREAD_LED_STACK_SIZE];
24.
25.  /* 线程做无用操作，起到空转效果 */
26.  static Delay(TWord count)
27.  {
28.      while(count--);
29.  }
30.
31.  /* 线程 LED1 的入口函数 */
32.  static void ThreadLED1Entry(void* pArg)
33.  {
34.      TByte turn = 0;
35.      while (eTrue)
36.      {
37.         EVB_LEDControl(LED_INDEX1, LED_ON);
38.         Delay(0xFFFFFF);
39.         EVB_LEDControl(LED_INDEX1, LED_OFF);
40.         Delay(0xFFFFFF);
41.         turn ++;
42.         if (turn == 2)
43.         {
44.             TclSetThreadPriority(&ThreadLED2, THREAD_LED_PRIORITY - 1);
45.             turn = 0;
46.         }
47.      }
48.  }
49.
50.  /* 线程 LED2 的入口函数 */
51.  static void ThreadLED2Entry(void* pArg)
52.  {
53.      TByte turn = 0;
54.      while (eTrue)
55.      {
56.         EVB_LEDControl(LED_INDEX2, LED_ON);
57.         Delay(0xFFFFFF);
58.         EVB_LEDControl(LED_INDEX2, LED_OFF);
59.         Delay(0xFFFFFF);
60.         turn ++;
61.         if (turn == 2)
62.      {
63.             TclSetThreadPriority(&ThreadLED2, THREAD_LED_PRIORITY + 1 );
64.             turn = 0;
65.    }
```

```
66.  }
67. }
68.
69.  /* 用户应用入口函数 */
70.  static void EVB_LEDAppEntry(void)
71.  {
72.      /* 配置使能评估板上的 LED 设备 */
73.      EVB_LEDConfig();
74.
75.      /* 初始化 LED1 设备控制线程 */
76.      TclInitThread(&ThreadLED1, &ThreadLED1Entry, (void*)NULL,
77.                    ThreadLED1Stack, THREAD_LED_STACK_SIZE,
78.                    THREAD_LED_PRIORITY, THREAD_LED_SLICE);
79.
80.      /* 初始化 LED2 设备控制线程 */
81.      TclInitThread(&ThreadLED2, &ThreadLED2Entry, (void*)NULL,
82.                    ThreadLED2Stack, THREAD_LED_STACK_SIZE,
83.                    THREAD_LED_PRIORITY + 1, THREAD_LED_SLICE);
84.
85.      /* 激活 LED1 设备控制线程 */
86.      TclActivateThread(&ThreadLED1);
87.
88.      /* 激活 LED2 设备控制线程 */
89.      TclActivateThread(&ThreadLED2);
90.  }
91.
92.  /* 处理器 BOOT 之后会调用 main 函数，必须提供 */
93.  int main(void)
94.  {
95.      /* 注册处理器初始化函数到内核 */
96.      TclSetCpuEntry(&TCL_STM32F10xInit);
97.
98.      /* 注册板级初始化函数到内核 */
99.      TclSetBoardEntry(&EVB_SetupBoard);
100.
101.     /* 注册板级调试打印函数到内核 */
102.     TclSetTraceRoutine(&EVB_Uart1WriteByte);
103.
104.     /* 注册用户初始化函数到内核 */
105.     TclSetUserEntry(&EVB_LEDAppEntry);
106.
107.     /* 启动内核 */
108.     TclStartKernel();
109.
110.     return 1;
111. }
112.
113.
114. #endif
```

程序运行结果如图 2-27 所示。

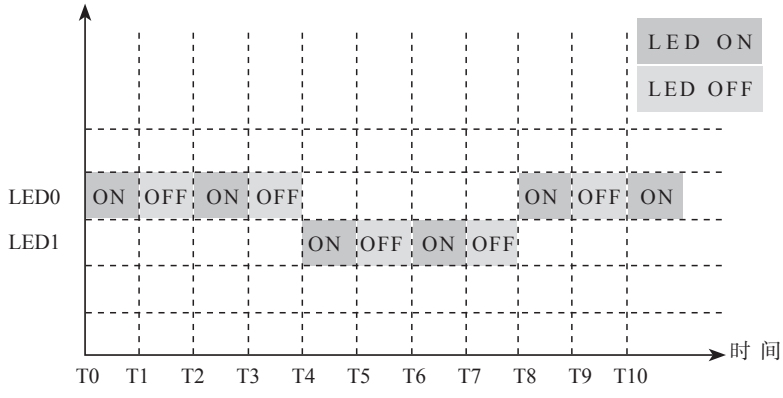

图 2-27 代码清单 2-18 运行结果

2.6.6 线程时间片修改演示

我们以 Colibri 评估板为例，下面的代码演示了如何使用该 API。我们的目标是在评估板上通过三个独立线程的时间片的变动，来实现 LED 亮 / 暗的不同占空比。也就是说 LED 的点亮时间和熄灭时间的长短比例是变化的。具体代码如代码清单 2-19 所示。

代码清单2-19：修改线程时间片

```
1.  #include "trochili.h"
2.  #include "example.h"
3.
4.  #if (EVB_EXAMPLE == CH2_THREAD_EXAMPLE7)
5.
6.  /* LED 设备参数 */
7.  #define LED_ON     (1)
8.  #define LED_OFF    (0)
9.  #define LED_INDEX (2)
10.
11. /* 创建线程时需要的参数 */
12. #define THREAD_LED_STACK_SIZE    (256*2)
13. #define THREAD_LED_PRIORITY      (5)
14. #define THREAD_LED_SLICE         (100)
15. #define THREAD_LED_SLICE_MAX     (400)
16. #define THREAD_CTRL_STACK_SIZE   (256*2)
17. #define THREAD_CTRL_PRIORITY     (5)
18. #define THREAD_CTRL_SLICE        (100)
19.
20. /* 线程定义 */
21. static TThread ThreadLEDOn;
22. static TThread ThreadLEDOff;
```

```
23.  static TThread ThreadCTRL;
24.
25.  /* 线程栈定义 */
26.  static TWord ThreadLEDOnStack[THREAD_LED_STACK_SIZE];
27.  static TWord ThreadLEDOffStack[THREAD_LED_STACK_SIZE];
28.  static TWord ThreadCTRLStack[THREAD_CTRL_STACK_SIZE];
29.
30.  /* 线程 LEDOn 的线程函数 */
31.  static void ThreadLEDOnEntry(void* pArg)
32.  {
33.      while (eTrue)
34.      {
35.          EVB_LEDControl(LED_INDEX, LED_ON);
36.      }
37.  }
38.
39.  /* 线程 LEDOff 的线程函数 */
40.  static void ThreadLEDOffEntry(void* pArg)
41.  {
42.      while (eTrue)
43.      {
44.          EVB_LEDControl(LED_INDEX, LED_OFF);
45.      }
46.  }
47.
48.  /* 线程 CTRL 的线程函数 */
49.  static void ThreadCTRLEntry(void* pArg)
50.  {
51.      static TTimerTick slice = THREAD_LED_SLICE;
52.      while (eTrue)
53.      {
54.          TclSetThreadSlice(&ThreadLEDOn, slice);
55.          if (slice < THREAD_LED_SLICE_MAX)
56.          {
57.              slice += THREAD_LED_SLICE;
58.          }
59.          else
60.          {
61.              slice = THREAD_LED_SLICE;
62.          }
63.
64.          TclYieldThread();
65.      }
66.  }
67.
68.  /* 用户应用入口函数 */
69.  static void EVB_LEDAppEntry(void)
70.  {
```

```
71.      /* 配置使能评估板上的 LED 设备 */
72.      EVB_LEDConfig();
73.
74.      /* 初始化 CTRL 线程 */
75.      TclInitThread(&ThreadCTRL, &ThreadCTRLEntry, (void*)NULL,
76.                    ThreadCTRLStack, THREAD_CTRL_STACK_SIZE,
77.                    THREAD_CTRL_PRIORITY, THREAD_CTRL_SLICE);
78.
79.      /* 初始化 LED 设备控制线程 */
80.      TclInitThread(&ThreadLEDOn, &ThreadLEDOnEntry, (void*)NULL,
81.                    ThreadLEDOnStack, THREAD_LED_STACK_SIZE,
82.                    THREAD_LED_PRIORITY, THREAD_LED_SLICE);
83.
84.      /* 初始化 LED 设备控制线程 */
85.      TclInitThread(&ThreadLEDOff, &ThreadLEDOffEntry, (void*)NULL,
86.                    ThreadLEDOffStack, THREAD_LED_STACK_SIZE,
87.                    THREAD_LED_PRIORITY, THREAD_LED_SLICE);
88.
89.      /* 激活 CTRL 线程 */
90.      TclActivateThread(&ThreadCTRL);
91.
92.      /* 激活 LED 设备控制线程 */
93.      TclActivateThread(&ThreadLEDOn);
94.
95.      /* 激活 LED 设备控制线程 */
96.      TclActivateThread(&ThreadLEDOff);
97.  }
98.
99.  /* 处理器 BOOT 之后会调用 main 函数，必须提供 */
100. int main(void)
101. {
102.     /* 注册处理器初始化函数到内核 */
103.     TclSetCpuEntry(&TCL_STM32F10xInit);
104.
105.     /* 注册板级初始化函数到内核 */
106.     TclSetBoardEntry(&EVB_SetupBoard);
107.
108.     /* 注册板级调试打印函数到内核 */
109.     TclSetTraceRoutine(&EVB_Uart1WriteByte);
110.
111.     /* 注册用户初始化函数到内核 */
112.     TclSetUserEntry(&EVB_LEDAppEntry);
113.
114.     /* 启动内核 */
115.     TclStartKernel();
116.
117.     return 1;
118. }
```

```
119.
120. #endif
```

可以通过图 2-28 的演示来理解上述代码，三个线程拥有相同的优先级，轮流执行，每次轮到控制线程运行时都会把 LEDOn 线程的时间片增加 1 秒，直到增加到指定的最大值后才返回 1 秒。

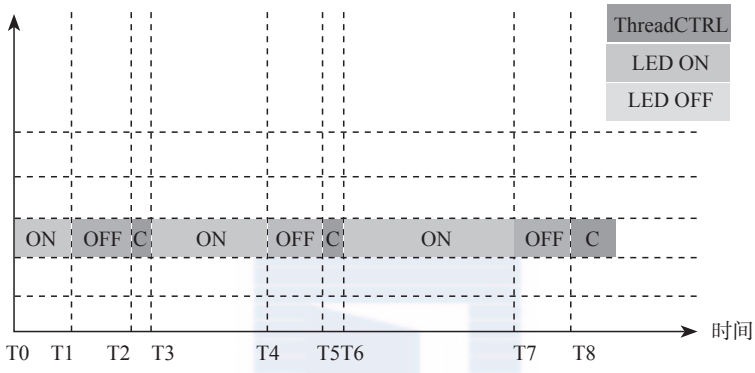

图 2-28 代码清单 2-19 图解

第 3 章
线程同步和通信

在嵌入式系统中运行的代码主要包括线程和 ISR，在它们的运行过程中，它们的运行步骤有时需要同步，它们的访问资源有时需要互斥，它们之间有时也要彼此交换数据。这些需求，有的是因为应用需求，有的是多任务编程模型带来的需求。因此操作系统必须提供相应的机制来完成这些功能。在这里把这些机制统称为进（线）程间通信（Internal Process Communication，IPC），常见的机制主要包括信号量、消息队列、邮箱、事件标记、管道、信号和条件变量等。

Trochili RTOS 目前支持的 IPC 机制包括信号量、互斥量、邮箱、消息队列和事件标记。在设计 IPC 机制时，在对这几种 IPC 机制仔细分析后，我发现在线程的阻塞和数据的传输流程上，有很多相似的地方，可以设计一套比较完整通用的代码作为 IPC 模块的基础。本章主要分析这些 IPC 机制的底层通用函数和数据结构。

3.1 线程阻塞队列

首先考虑一下线程队列的问题。前面有关线程的章节中，定义了两个内核线程队列：内核线程就绪队列和内核线程辅助队列。在 IPC 部分，我们又定义了另一种类型的线程队列：线程阻塞队列。各类型的 IPC 对象，均带有线程阻塞特性，需要保存那些因 IPC 操作失败而不得不等待 IPC 条件满足的线程，而线程阻塞队列正是用来实现这个功能的。

在 Trochili RTOS 中，IPC 线程阻塞队列定义在文件 ipc.h 中。和其他 RTOS 相比，这个线程阻塞队列实现有自己的特点：线程阻塞队列中设计了两个线程阻塞分队列。这样做的理由是像邮箱、消息队列的功能里，会把数据分为紧急和一般的消息 / 邮件，所以需要分别保存到不同的队列中。如果同时有紧急数据和普通数据到达，内核优先考虑的是紧急数据。而同样类型的数据，比如都是普通消息或者都是紧急消息，那就既可以按照 FIFO 机制来处理，也可以按照发送数据线程的优先级机制来处理。线程阻塞队列结构如代码清单 3-1 所示。

代码清单3-1：线程阻塞队列结构定义

```
1. /* IPC 线程阻塞队列结构定义 */
2. struct IPCBlockedQueueDef
3. {
```

```
4.      TProperty* Property;              /* 队列宿主属性参数指针 */
5.      TLinkNode* PrimaryHandle;         /* 队列中基本线程分队列 */
6.      TLinkNode* AuxiliaryHandle;       /* 队列中辅助线程分队列 */
7. };
```

- Property 宿主属性参数指针：指向具体 IPC 对象的属性字段，在属性字段中，会有分队列的调度策略的参数，指明两个分队列按照哪种方式来调度：FIFO 或者线程优先级。
- PrimaryHandle 队列中基本线程分队列：没有特殊要求的线程会阻塞在这个队列中。
- AuxiliaryHandle 队列中辅助线程分队列：在该队列中阻塞的线程会被特殊处理。

通过这几个成员，针对线程的优先级、数据的特性，用户可以对每个 IPC 对象做出最符合应用的配置。图 3-1 基于信号量的结构演示了 IPC 线程阻塞队列结构。

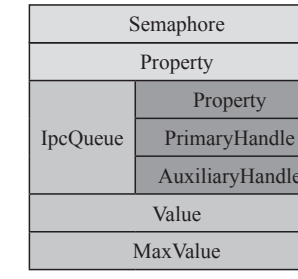

图 3-1 信号量的线程阻塞队列演示

3.2 线程阻塞记录

前面曾经提到过，在线程结构中有一个 IpcRecord 成员，当线程因操作 IPC 对象不能完成时，有时需要阻塞在 IPC 对象的线程阻塞队列上，而此时需要记录该线程在线程阻塞队列上的阻塞情况。包括被操作的 IPC 对象类型和地址，操作时的参数以及用于传递数据的指针。这些信息都保存在 IpcRecord 结构中。该成员的结构定义如代码清单 3-2 所示。

代码清单3-2：IPC阻塞记录结构定义

```
1. /* 线程用于记录 IPC 对象的详细信息的记录结构 */
2. struct IpcRecordDef
3. {
4.      void*       Resource;       /* 指向 IPC 对象地址的指针          */
5.      TIpcQueue*  Queue;          /* 线程所属 IPC 线程队列指针        */
6.      TIpcData    Data;           /* 和 IPC 对象操作相关的数据指针    */
7.      TOption     Option;         /* 访问 IPC 对象的操作参数          */
8.      TState      State;          /* IPC 对象操作的返回值             */
9.      TErrno      Errno;          /* IPC 对象操作的错误代码           */
10.};
11.typedef struct IpcRecordDef TIpcRecord;
```

- Resource：被操作的 IPC 对象的地址。
- Queue：线程所在的 IPC 的线程阻塞队列地址。
- Data：如果线程需要交互数据，则这个变量指向线程待传输数据的地址。
- Option：记录 IPC 操作的参数，比如是否有时限要求，是否在辅助阻塞队列中。
- State：线程操作 IPC 的结果，可能是成功、失败、被取消、被取消初始化等。
- Errno：IPC 对象操作的错误代码。

图 3-2 演示了线程访问信号量时的阻塞情况。

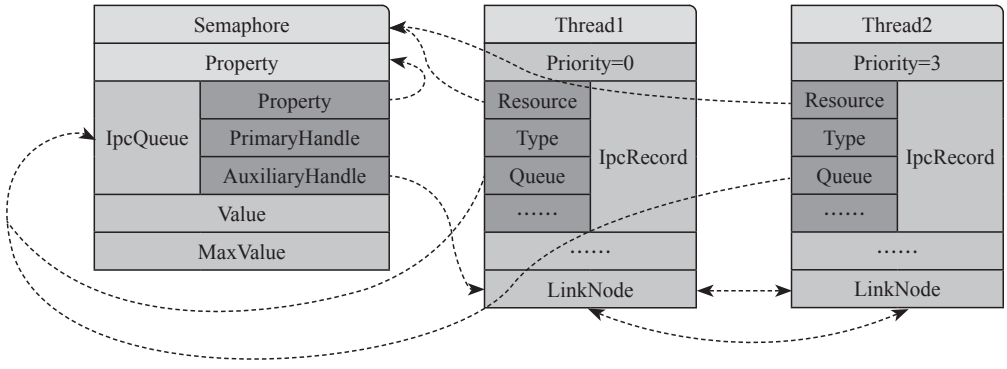

图 3-2　信号量的线程阻塞现场

从图 3-2 中我们可以看到：

- 信号量的结构中有一个线程阻塞队列。
- 线程阻塞队列的 Property 直接指向信号量的 Property 参数。
- 线程阻塞队列中的基本队列中有两个线程被阻塞。
- 被阻塞线程通过 IpcRecord 记录下当前操作的信号量的地址、阻塞队列的地址。所在阻塞队列的分队列由 IpcRecord 的 Option 成员记录。

3.3　IPC 机制底层支撑函数

当线程以阻塞方式访问一个 IPC 对象时，如果操作不能立刻成功，那么这个线程就要被阻塞了。访问每种类型的 IPC 对象的机制是不一样的，但阻塞时的流程却都差不多。需要几个函数配合来完成具体的阻塞操作。线程阻塞 – 解除阻塞的过程如下描述：

- 首先是线程发起 IPC 操作，但操作不能成功。
- 此时需要保存当前访问的 IPC 信息，如果是消息传递类型的 IPC 对象，还要保存被传递（发送和接收）的数据变量的地址。
- 然后当前线程的状态会被设置为阻塞态，并且会从内核线程就绪队列移入线程阻塞队列。如果需要，还会立刻发出线程调度。
- 当 IPC 对象满足那些被阻塞的线程时，就需要选择其中一个最恰当的线程解除阻塞，如果需要，还要把数据传给那个刚被解除阻塞的线程。
- 等到该线程再次被调度时，它会检查本次 IPC 操作的结果并随后清除 IpcRecord 记录。

本模块的函数基本都是 IPC 机制的内部支持函数，这些函数只会被各 IPC 功能的代码来调用，并不直接面向用户，也不会被 API 层直接调用。主要包括以下几个函数（见表 3-1）。

表 3-1 IPC 底层支持函数列表

编号	函数	作用
1	uIpcInitQueue	初始化线程阻塞队列
2	uIpcSaveRecord	保存 IPC 对象操作的信息到线程结构
3	uIpcCleanRecord	清除 IPC 对象操作的信息
4	uIpcReadState	获得 IPC 对象访问的结果
5	uIpcBlockThread	将当前线程阻塞在线程阻塞队列上
6	uIpcUnblockThread	将阻塞队列上的指定线程正常解除阻塞
7	uIpcUnblockOptimalThread	在阻塞队列上选择一个合适的线程并解除阻塞
8	uIpcUnblockAllThreads	将阻塞队列上的所有线程解除阻塞
9	uIpcAbortThread	将阻塞队列上的某个线程强制解除阻塞
10	uIpcDeActivate	将阻塞队列上的某个线程休眠
11	uIpcSetThreadPriority	修改阻塞队列上的某个线程的优先级

3.3.1 线程阻塞队列初始化

函数 uIpcInitQueue() 用来清空队列的两个分队列头，然后将队列的属性成员指向宿主对象的属性成员。这样在操作队列时，可以直接引用宿主对象的各种配置参数而不必再去查找宿主地址。

3.3.2 保存线程阻塞信息

当线程需要在 IPC 对象上阻塞时，首先会把线程操作的对象的信息保存在线程结构中的 IpcRecord 结构成员中。这是通过函数 uIpcSaveRecord() 完成的。

3.3.3 清除线程阻塞信息

线程读取阻塞操作的结果后，通过函数 uIpcCleanRecord() 清理 IpcRecord 记录，然后返回主调函数，至此 IPC 操作结束。

3.3.4 读取线程阻塞结果

当阻塞状态的线程被解除阻塞后，线程从阻塞点继续执行，首先会通过函数读 uIpcReadState() 取引起本次阻塞的操作的结果，操作结果记录在 IpcRecord 记录里。

3.3.5 线程阻塞过程

线程通过阻塞方式访问 IPC 对象，如果条件不满足则通过函数 uIpcBlockThread() 将自己阻塞。主要通过以下步骤来完成阻塞操作：

- ❏ 保存本次操作 IPC 的信息到 IpcRecord 中。
- ❏ 将自己从内核线程就绪队列中移出。
- ❏ 将自己加入到 IPC 对象的线程阻塞队列。

❑ 修改自己的状态为阻塞态。
❑ 如果是时限阻塞方式，则将启动自己的线程定时器。

3.3.6 解除线程阻塞过程

内核通过函数 uIpcUnblockThread() 解除指定线程的阻塞。主要通过以下步骤来完成操作：

❑ 将线程从 IPC 对象的线程阻塞队列中移出。
❑ 将线程加入到内核线程就绪队列中。
❑ 修改线程的状态为就绪态。
❑ 如果是时限阻塞方式，则停止线程的私有定时器。

3.3.7 解除最佳线程阻塞过程

内核通过函数 uIpcUnblockOptimalThread() 从 IPC 对象的线程阻塞队列中选择最恰当的线程并解除它的阻塞。主要通过以下步骤来完成操作：

❑ 查找 IPC 阻塞队列中的最佳线程 , 辅助队列中的线程优先考虑。
❑ 通过 uIpcUnblockThread() 函数来完成最终的阻塞解除操作。

3.3.8 解除全部线程阻塞过程

内核通过函数 uIpcUnblockAllThreads() 逐一将 IPC 对象的线程阻塞队列的线程解除阻塞：

❑ 循环检查 IPC 对象的线程阻塞队列的辅助队列，通过 uIpcUnblockThread() 函数将其中的全部线程逐一解除阻塞。
❑ 循环检查 IPC 对象的线程阻塞队列的基本队列，通过 uIpcUnblockThread() 函数将其中的全部线程逐一解除阻塞。

3.3.9 强制解除线程阻塞

函数 uIpcAbortThread() 可以用来强制解除某个线程在 IPC 对象上的阻塞。和以上函数不同，本函数是一种强制操作，并没有按照 IPC 对象阻塞方案的流程来处理。函数首先检查被解除阻塞的线程是否阻塞在指定的 IPC 对象上，如果是则通过函数 uIpcUnblockThread() 来解除线程的阻塞。

3.3.10 休眠被阻塞的线程

内核通过函数 uIpcDeActivate() 解除指定线程的阻塞，然后将线程休眠（不是置为就绪）。主要有以下步骤来完成操作：

❑ 将线程从 IPC 对象的线程阻塞队列中移出。

- 将线程加入到内核线程辅助队列中。
- 修改线程的状态为休眠态。
- 如果是时限阻塞方式，则停止线程的私有定时器。

3.3.11 设置被阻塞线程的优先级

函数 uIpcSetThreadPriority() 用来改变处在 IPC 阻塞队列中的线程的优先级。

- 如果线程所属队列采用优先级策略，则将线程从所属的资源阻塞队列中移出，然后修改它的优先级 , 最后再放回原队列。
- 如果是先入先出队列则直接修改线程优先级。

以上介绍的这些函数是 Trochili RTOS IPC 的底层通用机制，内核通过这些最小功能的函数可以支持各种具体 IPC 对象的功能的实现。随后的章节将详细介绍如何实现各种 IPC 机制。