第4章

RTOS构建块

系统开发

介绍

RTOS必须为实时嵌入式系统的开发人员提供各种服务。这些服务允许开发人员创建、操作和管理系统资源和实体，以促进应用程序的开发。本章的主要目标是回顾在Azure RTOS ThreadX中可用的服务和组件。图19包含了这些服务和组件的摘要。

线程	消息队列	计数 半信号器
穆德塞斯	事件标志	内存块池
内存字节池	应用程序计时器	计时器和中断控制

图19。Azure RTOS ThreadX组件

定义公共资源

所讨论的一些组成部分被表示为公共资源。如果一个组件是一个公共资源，则意味着可以从任何线程访问它。请注意，访问一个组件与拥有该组件是不一样的。例如，一个互斥锁可以从任何线程访问，但它一次只能由一个线程拥有。

Azure RTOS的ThreadX数据类型

Azure RTOS ThreadX使用特殊的原始数据类型，它们直接映射到底层C编译器的数据类型。这样做是为了确保在不同的C编译器之间的可移植性。图20包含了Azure RTOS ThradX服务调用数据类型及其相关含义的摘要。

数据类型	描述
尤因特	基本无符号整数。此类型必须支持8位无符号数据；但是，它被映射到最方便的无符号数据类型，该类型可能支持16位或32位有符号数据。
乌龙	无符号长类型。此类型必须支持32位无符号数据。
无效	几乎总是等同于编译器的空白类型。
焦油	通常是一个标准的8位字符类型。

图20。Azure RTOS ThreadX基本体数据类型

除了原始数据类型外，Azure RTOS ThradX还使用系统数据类型来定义和声明系统资源，例如线程和互变量。图21包含了这些数据类型的摘要。

系统数据类型	系统资源
tx_timer	应用程序计时器
tx_queue	消息队列
tx_thread	应用程序线程
tx_semaphore	计数信号量
tx_event_flags_group	事件标志组
tx_block_pool	内存块池
tx_byte_pool	内存字节池
tx_mutex	麦克斯

图21。Azure RTOS ThreadX系统数据类型

螺纹

一个线程是一个半独立的程序段。进程中的线程共享相同的内存空间，但是每个线程必须有自己的堆栈。线程是基本的构建块，因为它们包含了大部分的应用程序编程逻辑。有

对可以创建的线程的数量没有明确的限制，并且每个线程可以有不同的堆栈大小。当执行线程时，它们会相互独立地进行处理。

图22。线程的属性

当创建一个线程时，需要指定几个属性，如图22所示。每个线程都必须有一个线程控制块（TCB），其中包含对该线程的内部处理至关重要的系统信息。但是，大多数应用程序都不需要访问TCB的内容。每个线程都被分配了一个名称，它主要用于识别目的。线程输入函数是一个线程的实际C代码所在的位置。线程输入输入是在第一次执行时传递给线程输入函数的值。线程条目输入值的使用完全由开发人员决定。每个线程都必须有一个堆栈，因此指定一个指向实际堆栈位置的指针以及堆栈大小。必须指定线程优先级，但它可以在运行时进行更改。抢占阈值是一个可选值；等于优先级的值将禁用抢占阈值功能。可以分配一个可选的时间片，它指定在允许其他具有相同优先级的已就绪线程运行之前允许该线程执行的计时器标记数。请注意，使用抢占阈值会禁用时间片选项。时间切片值为0(0)将禁用此线程的时间切片。最后，必须指定一个启动选项，该选项指示线程是立即启动，还是处于挂起状态，必须等待另一个线程激活它。

内存池

一些资源在创建这些资源时需要分配内存空间。例如，在创建一个线程时，必须为其堆栈提供内存空间。Azure RTOS ThreadX提供了两种内存管理技术。开发人员

可以选择其中一种技术进行内存分配，或任何其他分配内存空间的方法。

第一种内存管理技术是内存字节池，如图23所示。顾名思义，内存字节池是可用于任何资源的字节的顺序集合。内存字节池类似于标准的C堆。与C堆不同，对内存字节池的数量没有限制。此外，线程可以挂起池，直到所请求的内存可用。来自内存字节池的分配是基于指定的字节数进行的。Azure RTOS ThreadX以第一个拟合的方式从字节池中分配，i。e.，将使用满足该请求的第一个可用内存块。*此块中的多余内存被转换为新块并放回空闲内存列表中，通常会导致碎片化*。Azure RTOS ThreadX在后续的分配搜索过程中将相邻的空闲内存块合并在一起。*这个过程被称为碎片整理*。

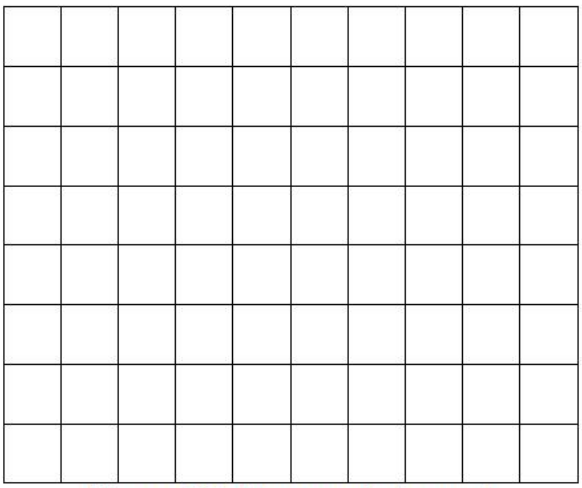


Figure 23. Memory byte pool

图24包含了一个内存字节池的属性。每个内存字节池都必须有一个包含基本系统信息的控制块。每个内存字节池都被分配了一个名称，它主要用于用户识别目的。必须提供字节池的起始地址，以及要分配给内存字节池的字节总数。

第二种类型的内存管理技术是内存块池，如图25所示。内存块池由固定大小的内存块组成，因此永远不会出现碎片化问题。由于每次都分配了相同数量的内存，因此缺乏灵活性。但是，对于可以创建多少内存块池没有限制，而且每个池可以有不同的内存块大小。通常，内存块池比内存字节池更可取，因为碎片问题，而且对池的访问速度更快。

内存字节池控制块
内存字节池名称
字节池的位置
已分配的字节总数

图24。内存字节池的属性

图26包含了一个内存块池的属性。每个内存块池都必须有一个包含重要系统信息的控制块。每个内存块池都分配一个名称，主要用于识别目的。必须指定每个固定大小的内存块中的字节数。必须提供内存块池所在的地址。最后，必须指示对整个内存块池可用的总字节数。

固定尺寸的块
固定尺寸的块
固定尺寸的块
:
固定尺寸的块

图25。内存块池

内存块存储器池控制块
内存块池名称
每个内存块中的字节数
内存块池的位置
可用的字节总数

图26。内存块池的属性

内存块池中的内存块总数可以计算如下：

$$\text{方块总数} = \frac{\text{合计 数量 的 字节 可用}}{(\text{每个内存块中的字节数}) + (\text{大小}(\texttt{void*}))}$$

每个内存块包含一个用户不可见的开销指针，它由前面公式中的sizeof（void*）表达式表示。通过根据所需的内存块数正确计算要分配的字节总数，来避免浪费内存空间。

应用程序计时器

快速响应异步的外部事件是实时、嵌入式应用程序最重要的功能。然而，许多这些应用程序还必须在预定的时间间隔内执行某些活动。应用程序计时器允许应用程序在特定的时间间隔内执行应用程序C功能。一个应用程序计时器也有可能只过期一次。这种类型的计时器被称为一次性计时器，而重复的间隔计时器被称为周期计时器。每个应用程序计时器都是一个公共资源。

应用计时器控制块
应用程序计时器名称
调用的过期函数
要传递给函数的过期输入值
计时器的初始数量
重新安排定时器标记的数量
自动激活选项

图27。应用程序计时器的属性

图27包含了一个应用程序计时器的属性。每个应用程序计时器都必须有一个包含基本系统信息的控制块。每个应用程序计时器都被分配了一个名称，它主要用于识别目的。其他属性包括在计时器过期时执行的过期函数的名称。另一个属性是传递给过期函数的值。（此值仅供开发人员使用。）一个包含初始计时器标记的数量的属性[17]对于计时器过期是必需的，还有一个属性指定第一个计时器过期后所有计时器过期的计时器点数。最后一个属性用于指定应用程序计时器是否在创建时被自动激活，或者它是否被创建为需要一个线程来启动它的非活动状态。

[17]计时器计时器之间的实际时间由应用程序指定，但10 ms是这里使用的值。

应用程序计时器与isr非常相似，除了实际的硬件实现（通常使用一个周期性的硬件中断）被隐藏在应用程序中。这些计时器被应用程序用来执行超时、定期操作和/或监督服务。就像isr一样，应用程序计时器最经常中断线程的执行。然而，与isr不同的是，应用程序计时器不能相互中断。

麦克斯

互斥锁的唯一目的是提供互斥；这个概念的名称提供了互斥锁名的派生(i。e.，多次排除）。[18]互斥锁用于控制线程对关键部分或某些应用程序资源的访问。互斥锁是一种只能由一个线程拥有的公共资源。可以定义的互变量的数量没有限制。图28包含了一个互斥锁的属性的摘要。

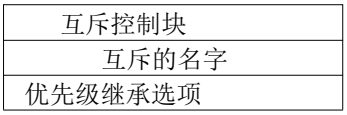


图28。互斥的属性

每个互斥锁都必须有一个包含重要系统信息的控制块。每个互斥锁都被分配了一个名称，这主要用于识别目的。第三个属性指示该互斥锁是否支持优先级继承。优先级继承允许低优先级线程暂时承担高优先级线程的优先级，该线程正在等待由低优先级线程拥有的互斥锁。该功能通过消除对中间线程优先级的抢占，帮助应用程序避免不确定性优先级反转。互斥锁是唯一支持优先级继承的Azure RTOS ThreadX资源。

计数信号量

计数信号量是一种公共资源。没有符号所有权的概念，就像互音的情况一样。计数信号量的主要目的

[18]在20世纪60年代，埃德杰·迪克斯特拉提出了具有两种操作的互斥信号的概念：P操作（Prolaag，意思更低）和V操作（维根，意思提高）。如果其值大于零，则P操作将减少信号量，而V操作将增加信号量值。P和V是原子运算。

是事件通知、线程同步和互斥。[19]Azure RTOS ThreadX提供了32位计数信号量，其中计数必须在0到4、294、967、295或2的范围内^{32}-1（包括）。当创建计数信号量时，必须将计数初始化为该范围内的值。信号量中的每个值都是该信号量的一个实例。因此，如果信号量计数为5，那么该信号量就有5个实例。

图29包含了一个计数信号量的属性。每个计数信号量都必须有一个包含基本系统信息的控制块。每个计数信号量都被分配了一个名称，这主要用于识别目的。每个计数信号量都必须有一个指示可用实例数的信号量计数。如上所述，计数的值必须在从0x000000000到0x FFFFFFFF（包括）的范围内。计数信号量可以在初始化期间或在运行时期间由线程创建。可以创建的计数信号量的数量没有限制。

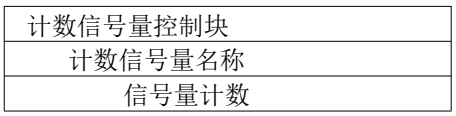
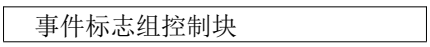


图29。计数信号量的属性

事件标志组

一个事件标志组是一个公共资源。事件标志为线程同步提供了一个强大的工具。每个事件标志用一个位表示，事件标志被分成32组排列。当创建事件标志组时，所有事件标志都将初始化为零。

图30包含了一个事件标志组的属性。每个事件标志组都必须有一个包含基本系统信息的控制块。每个事件标志组都被分配了一个名称，它主要用于标识目的。还必须有一组32个一位的事件标志，它位于控制块中。



[19] 在这种情况下，互斥通常是通过使用二进制信号量来实现的，这是计数信号量的一种特殊情况，其中计数被限制在值0和1。

事件标志组名称
由32个一位事件标志组成的一组

图30。事件标志组的属性

事件标志为线程同步提供了一个强大的工具。线程可以同时操作所有32个事件标志。事件标志组可以在初始化期间或由线程在运行期间创建。图31包含了一个事件标志组被初始化后的说明。可以创建的事件标志组的数量没有限制。

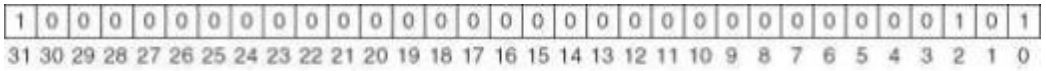


图31。事件标志组

消息队列

消息队列是一个公共资源。消息队列是线程间通信的主要手段。一个或多个消息可以驻留在一个消息队列中。包含单个消息的消息队列通常称为邮箱。消息被放置在队列的后面，[20]并被从队列的前面移除。

消息队列控制块
消息队列名称
每个消息的大小
消息队列的位置
消息队列的总大小

图32。消息队列的属性

图32包含了一个消息队列的属性。每个消息队列都必须有一个包含基本系统信息的控制块。每个消息队列分配一个名称，主要用于标识目的。其他属性包括消息大小、消息队列所在的地址以及分配给消息队列的总字节数。如果分配给的字节总数

[20] 也可以在队列的前面插入一条消息。

消息队列不能被消息大小均匀地整除，则不使用剩余的字节。

图33包含了一个消息队列的说明。任何线程都可以在队列中插入消息（如果空间可用），并且任何线程都可以从队列中删除消息。

图33。消息队列

	麦克	计数信号量
速度	比a慢一些 臂板信号系统	信号量通常比互斥锁更快，并且需要更少的系统资源
线程所有权	只有一个线程可以拥有a 麦克斯	没有信号量的线程所有权概念——如果当前计数超过零，任何线程都可以减少计数信号量
优先级继承	只能使用互斥锁	功能不可用于 臂板信号系统
相互排斥	互斥体的主要目的-互斥体应该只用于互斥	可以通过使用二进制信号量来完成，但也可能存在陷阱
线程间同步	不要为此使用互斥锁 意图	可以使用信号量执行，但也应该考虑事件标志组
事件通知	不要为此使用互斥锁 意图	可以使用一个信号量来执行吗
螺纹悬挂	如果另一个线程已经拥有互斥锁（取决于等待选项的值），则线程可以挂起	如果计数信号量的值为零（取决于等待选项的值），则线程可以暂停

图34。互斥锁与计数信号量的比较

线程同步和通信组件的总结

互斥锁和计数信号量之间存在相似性，特别是在实现互斥时。特别是，一个二进制信号量有许多相同的特性

如互斥锁的属性。图34包含了这两种资源和关于应该如何使用每种资源的建议的比较。

我们讨论了一个线程可以用于各种目的的四种公共资源，以及每种情况都可能有用的四种情况。图35包含了对这些资源的推荐使用情况的摘要。

	线程同步	事件通知	相互排斥	线程间通信
麦克斯			首选	
计数信号量	好的，最好是参加一个活动	首选	好的	
事件标志组	首选	好的		
消息队列	好的	好的		首选

图35。资源的推荐使用

主要术语和短语

应用程序计时器二进制信号量控制块计数信号量碎片整理条目函数事件标志组事件通知首次拟合分配碎片堆信息贮存与检索邮筒内存块池内存字节池消息队列互斥权互斥现象单次计时器周期定时器先买权优先购买权阈值原始数据类型优先优先权继承性共享资源业务通话垛系统数据类型线螺纹悬挂线程同步

互斥锁监视器计时器

问题

1. 解释为什么将特殊的原语数据类型UINT、ULONG、VOID和CHAR用于服务调用，而不是使用标准的C原语数据类型。

2. 线程输入功能的目的是什么？

3. 在什么情况下，您会使用二进制信号量而不是互斥锁来进行互斥？

4. 一个线程只能拥有一个公共资源。这是哪种资源？

5. 假设您可以选择使用内存字节池或内存块池。你应该选择哪一个？证明你的答案是正确的。

6. 获取一个计数信号量的实例意味着什么？

7. 事件标志组可表示的数字组合的最大数目是多少？

8. 消息通常被添加到消息队列的后面。为什么要在消息队列的前面添加一条消息？

9. 讨论一次性计时器和周期计时器之间的区别。

10. 什么是计时器？

第5章

线程-必不可少的组件

介绍

您已经在前几章中研究了线程的几个方面，包括它们的目的、创建、组合和使用。在本章中，您将探讨所有直接影响线程的服务。要开始，您将检查线程控制块的目的和内容。您还将检查每个线程服务，重点介绍每个服务的特性和功能。

螺纹控制块

螺纹控制块（TCB）[21]是一种用于在运行时期间维护线程状态的结构。在需要进行上下文切换时，它还用于保存有关线程的信息。图36包含了组成TCB的许多字段。

字段描述	
tx_thread_id	控制块 身份证
tx_run_count	线程的运行 柜台
tx_stack_ptr	线程的堆栈指 针
tx_stack_start	堆栈起始地址

“现场的描述》	
tx_state	线程的执行状态
tx_delayed_suspend	延迟 挂起标志
tx_suspending	螺纹 悬 旗
tx_preempt_threshold	优先购买权 阈值

[21]每个线程的特性都包含了tx_api。h文件。 在其TCB。此结构定义在

tx_stack_end	堆栈结束 地址
tx_stack_size	堆栈大小
tx_time_slice	当前时间 薄片
tx_new_time_slice	新的时间段
* tx_ready_next	指向 下一个准备就绪 线
* tx_ready_previous	指向上一个已 准备好的线程 的指针
tx_thread_name	指向线程名称 的指针
tx_priority	优先级 螺纹（0-31）

tx_priority_bit	优先级ID位
* tx_thread_entry	螺纹功 能点 进入
tx_entry_parameter	螺纹 功能 参数
tx_thread_timer	线程计时器 块
* tx_suspend_cleanup	螺纹 清扫 函数和 联系 数据
* tx_created_next	指向创建列表 中下一个线程 的指针
* tx_created_previous	指向创建列表 中前一个线程 的指针

图36。螺纹控制块

TCB可以位于内存中的任何位置，但最常见的方法是通过在任何函数的范围之外定义控制块而使其成为全局结构。[22]在其他区域定位控制块需要更加小心，就像所有动态分配的内存一样。如果在C函数中分配了一个控制块，那么与之相关联的内存将被分配到调用线程的堆栈上。通常，请避免使用本地存储作为控制块，因为一旦函数返回，就会释放它的整个本地变量堆栈空间——而不管另一个线程是否将其用于控制块。

在大多数情况下，开发人员不需要知道TCB的内容。然而，在某些情况下，特别是在调试过程中，检查某些字段（或成员）就会成为

[22] 关于TCB的存储和使用的注释也适用于其他Azure RTOS ThreadX实体的控制块。

很有用图37图37。线程控制块的两个有用成员包含有关对开发人员的两个更有用的TCB字段的详细信息。

tx_run_count这个成员包含多少
线程已被安排。递增计数器表示正在调度和执行该线程。

tx_state 此成员包含关联线程的状态。下面的列表表示可能的线程状态：

TX_READY 0x00
TX_COMPLETED 0x01
TX_TERMINATED 0x02
TX_SUSPENDED 0x03
TX_SLEEP 0x04
TX_QUEUE_SUSP 0x05
TX_SEMAPHORE_SUSP 0x06
TX_EVENT_FLAG 0x07
TX_BLOCK_MEMORY 0x08
TX_BYTE_MEMORY 0x09
TX_MUTEX_SUSP 0x0D
TX_IO_DRIVER 0x0A

图37。线程控制块的两个有用成员

TCB中还有许多其他有用的字段，包括堆栈指针、时间片值和优先级。开发人员可以检查TCB的成员，但严禁对其进行修改。没有显式值表示线程当前是否正在执行。在给定的时间内只执行一个线程，并且Azure RTOS ThreadX会在其他地方跟踪当前正在执行的线程。请注意，对于正在执行的线程的tx_state值为TX_READY。

线程服务摘要

附录A到J形成一个Azure RTOS ThreadX用户指南。这10个附录中的每一个都是专门针对一个特定的Azure RTOS ThreadX服务的。附录H包含关于线程服务的详细信息，包括每个服务的以下项目：原型、服务的简要描述、参数、返回值、注释和警告、允许调用、抢占可能性，以及说明的示例

可以使用服务。图38包含了所有可用的线程服务的列表。在本章的以下部分中，我们将研究这些服务。我们将考虑这些服务的许多特性，并开发几个说明性的例子。

线程服务	描述
tx_thread_create	创建应用程序线程
tx_thread_delete	删除应用程序线程
tx_thread_identify	检索指向当前正在执行的线程的指针
tx_thread_info_get	检索有关线程的信息
tx_thread_preemption_change	更改应用程序线程的抢占阈值
tx_thread_priority_change	更改应用程序线程的优先级
tx_thread_relinquish	将控制移交给其他应用程序线程
tx_thread_resume	恢复已挂起的应用程序线程
tx_thread_sleep	为指定的时间暂停当前线程
tx_thread_suspend	挂起应用程序线程
tx_thread_terminate	终止应用程序线程
tx_thread_time_slice_change	更改应用程序线程的时间片
tx_thread_wait_abort	中止指定螺纹

图38。线程服务

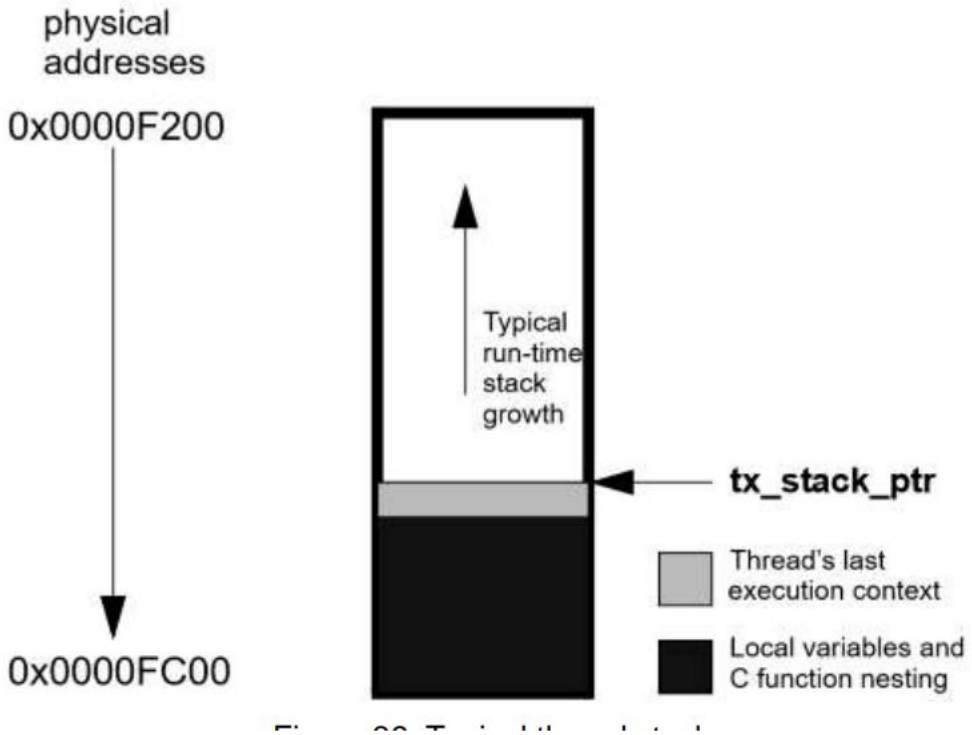
线程创建

使用TX_THREAD数据类型声明了一个线程[23]，并由tx_thread_create服务进行定义。每个线程必须有自己的堆栈；开发人员确定堆栈大小和为堆栈分配内存的方式。图39说明了一个典型的线程堆栈。有几种为堆栈分配内存的方法，包括使用字节池、块池和数组；或者只是在内存中指定一个物理起始地址。堆栈大小至关重要；它必须足够大，以适应最坏情况下的函数调用嵌套、局部变量分配和保存线程的最后一个执行上下文。预定义的最小堆栈大小常数TX_MINIMUM_STACK对于大多数应用程序来说可能太小了。错误大的堆栈比小的堆栈好。

[23] 当声明一个线程时，将创建一个线程控制块。

在开发人员调试完应用程序后，他/她可以微调堆栈，试图减少其大小。确定所需的堆栈空间的一种技术是在创建线程之前，用容易识别的数据模式预置所有堆栈区域，例如0xEFEF。在彻底测试了应用程序之后，您可以通过找到堆栈中预设模式仍然完整的区域来推断出实际使用了多少空间。

线程可能需要相当大的堆栈。因此，设计创建合理数量线程并避免过多堆栈的应用程序是很重要的
线程内的用法。开发人员通常应该避免递归算法和大型本地数据结构。



当一个堆栈区域太小时会发生什么？在大多数情况下，运行时环境只是假设有足够的堆栈空间。这将导致线程执行损坏其堆栈区域附近（通常是之前）的内存。结果是非常不可预测的，但大多数通常包括程序计数器的不自然变化。这通常被称为

跳进杂草中。当然，防止这个问题的唯一方法是确保所有的线程堆栈都足够大。

多线程的一个重要特性是，可以从多个线程调用相同的C函数。该特性提供了相当大的通用性，还有助于减少代码空间。但是，它确实要求从多个线程调用的C函数可以重入。重入函数是在已经执行时安全调用的函数。例如，如果函数由当前线程执行，然后由抢占线程再次调用，则会发生这种情况。[24]为了实现重入性，一个函数将调用者的返回地址存储在当前堆栈上（而不是存储在一个寄存器中），而不依赖于它之前设置的全局或静态C变量。大多数编译器确实会将返回地址放在堆栈上。因此，应用程序开发人员只需要担心全局参数和静态参数的使用。

非重入函数的一个例子是在标准C库中找到的字符串标记函数strtok。此函数通过将指针保存在静态变量中，在后续调用时记住前一个字符串指针。如果从多个线程调用这个函数，它很可能会返回一个无效的指针。

第4章说明了在Azure RTOS ThreadX中可用的各种构建块，包括线程属性。为了方便起见，在图40中再次说明了线程的属性。我们将使用tx_thread_create服务来创建几个线程来说明这些属性。

螺纹控制块	线程输入功能
线程名称	
线程输入	
堆栈（指针和大小）	
优先	
优先购买权阈值	
时间切片	
启动任选项	

图40。线程的属性

对于第一个线程创建示例，我们将创建一个优先级为15的线程，其入口点是“my_thed_entry”。这个线程的堆栈区域大小为1000字节，从

[24] 如果递归调用函数，也会发生这种情况。

地址0x400000。我们将不会使用抢占阈值特性，并且我们将禁用时间切片。一旦创建了线程，我们将使线程处于就绪状态。我们还需要创建一个线程输入函数来完成这个示例。图41包含了创建线程所需的代码及其相应的输入函数。

```
TX_THREAD我的线程;
UINT状态;

/*创建一个优先级为15的线程，其入口点为
    “我的线程条目”。这个线程的堆栈区域大小为1000字节，从地址0x400000开
    始。抢占阈值被设置为等于线程优先级，以禁用抢占阈值特性。禁用时间切片
    。这个线程将自动置入就绪状态。*/

状态=tx_thread_create(&我的线程，“我的线程”，我的线程条目，
                                    0x1234，
                                    （无效*）0x400000、1000、15、
                                    15、TX_NO_TIME_SLICE、
                                    TX_AUTO_START);

/*如果状态等于TX_SUCCESS，则my_thaed准备就绪
    执行*/

...

/*线程条目功能-当“my_线程”开始时
    执行时，控制被转移到这个函数*/

无效我的线程输入（ULONG初始输入）
{

/*线程的真正工作，包括调用
    其他函数应该在这里完成，*/

}
```

图41。创建一个优先级为15的线程

在这个图中，这个线

TX_THREAD我的线程;

用于定义一个名为my_thead的线程。回想一下，TX_THREAD是一种所使用的数据类型
定义TCB。这条线

UINT状态；

声明一个变量来存储来自服务调用调用的返回值。每次我们调用一个服务调用时，我们都将检查此状态变量的值，以确定该调用是否成功。我们将将此约定用于所有调用服务
调用，而不仅仅是为线程服务。开始的行

状态=tx_thread_create（……）；

创建线程，其中的参数指定了线程的特征。图42包含了对这些参数的描述。

参数	描述
&我的线程	指向线程控制块的指针（由TX_THREAD定义）
“我的线程”	指向线程名称的指针——一个用户定义的名称
my_tront_entry	线程输入函数的名称；当线程开始执行时，控制将传递给此函数
0x1234	传递给线程输入函数的32位值-此值保留以供应用程序单独使用
（语音*） 0x400000	堆栈内存区域的起始地址；我们使用了一个作为堆栈的起始位置的实际地址，尽管我们对如何分配堆栈空间有很多选择
1000	堆栈内存区域中的字节数
15	优先级-必须指定0到31（包括）的值
15	抢占阈值-等于优先级的值禁用抢占阈值
tx_no_time_slice	时间切片选项-这意味着我们已经禁用了这个线程的时间切片
tx_auto_start	初始线程状态-这意味着线程在创建后立即启动

图42。线程创建上一图中使用的参数

我们需要为这个线程创建一个线程输入函数。在这种情况下，是线

```
无效我的线程输入（ULONG初始输入）
{
…
}
```

定义该函数。如前所述，线程的实际工作，包括对其他函数的调用，都会在这个函数中发生。初始值_input值被传递给函数，并且只由应用程序使用。许多入口函数都处于“永远做”循环中，并且永远不会返回，但是如果函数确实返回，那么线程将被放置为“完成”状态。如果一个线程处于此状态，则不能再次执行它。

请参考附录，找到所有服务调用参数的详细描述，以及指示调用是否成功的返回值，如果成功，问题的确切原因。

*对于下一个线程创建示例，我们将创建一个优先级为20的线程，也具有“my_shoed_entry”的入口点。这个线程的堆栈区域大小为1500字节，从地址和my_stack开始。*我们将使用抢占阈值14，并将禁用时间切片。请注意，使用抢占阈值会自动禁用时间切片。抢占阈值为14意味着该线程只能被优先级高于14的线程抢占，即优先级从0到13（包括）。图43包含了创建此线程所需的代码。

```
TX_THREAD我的线程;
UINT状态;

/*创建一个优先级为20的线程，其入口点是“my_thed_entry”。这个线程的堆栈区域大小
  为1500字节，从地址和my_stack开始。抢占优先阈值是允许在优先级上抢占。时间切片
  已被禁用。这个线程将自动置入就绪状态。*/

状态=tx_thread_create（&我的线程，“我的线程”，我的线程条目，0x1234，和我的堆栈
                ，1500,20,14，TX_NO_TIME_SLICE，TX_AUTO_START）;

/*如果状态等于TX_SUCCESS，my_tled准备执行*/
```

图43。创建具有优先级= 20和优先抢占阈值= 14的线程

*对于我们最后的线程创建示例，我们将创建一个优先级为18的线程，同样带有一个“my_slaed_entry”的入口点和一个从&my_stack开始的堆栈。*此线程的堆栈

面积大小为1000字节。我们将不会使用抢占阈值，但我们将使用一个值为100个计时器标记的时间片值。图44包含了创建此线程所需的代码。请注意，时间切片确实会导致少量的系统开销。它仅在多个线程共享相同的优先级的情况下才有用。如果线程具有唯一的优先级，则不应使用时间切片。

```
TX_THREAD我的线程;
UINT状态;

/*创建一个优先级为18的线程，其入口点是“my_thed_entry”。这个线程的堆栈区
  域大小为1000字节，从地址&my_stack开始。已禁用抢占阈值功能。时间片是100
  个计时器点。
 这个线程将自动置入就绪状态。*/

状态=tx_thread_create（&我的线程，“我的线程”，我的线程条目，
                        0x1234，和我的堆栈，1000,18,18,100，
                        TX_AUTO_START）;

/*如果状态等于TX_SUCCESS，my_tled准备执行*/
```

图44。创建一个优先级为18且没有抢占阈值的线程

线程创建有八个可能的返回值，但只有一个表示线程创建成功。确保您在每次服务呼叫后检查返回状态。

线程删除

只有当一个线程处于已终止或已完成的状态时，才能删除它。因此，无法从试图删除自身的线程中调用此服务。通常，此服务是由计时器或其他线程调用。图45包含一个显示如何删除线程my_thaed的示例。

```
TX_THREAD我的线程;
UINT状态;
…
/*删除通过调用来创建其控制块的应用程序线程
  tx_thread_create .*/

状态=tx_thread_delete（&my_thaed）;
/*如果状态为TX_SUCCESS，则已删除该应用程序线程。*/
```

图45。*删除线程我的线程*

应用程序有责任管理被删除的线程堆栈所使用的内存区域，该内存区域在线程被删除后就可用了。此外，应用程序必须在一个线程被删除后防止使用它。

识别线程

tx_thread_identify服务返回一个指向当前正在执行的线程的指针。如果没有线程正在执行，则此服务将返回一个空指针。下面是一个显示如何使用此服务的示例。

```
my_thread_ptr = tx_thread_identify() ;
```

如果这个服务是从ISR中调用的，那么返回值表示在执行中断处理程序之前运行的线程。

获取线程信息

所有的Azure RTOS ThreadX对象都有三个服务，使您能够检索有关该对象的重要信息。线程的第一个服务——tx_thread_info_get服务—从线程控制块检索信息子集。此信息提供了在特定时刻，即调用服务时的“快照”。另外两个服务提供了基于运行时性能数据的收集的汇总信息。其中一个服务——tx_thread_performance_info_get服务——在调用服务之前为特定线程提供信息摘要。相比之下，tx_thread_performance_system_info_get会检索到调用服务之前系统中所有线程的信息摘要。这些服务对于分析系统的行为和确定是否存在潜在的问题领域很有用。tx_thread_info_get[25]服务获得的信息包括线程的当前执行状态、运行计数、优先级、抢占阈值、时间片、指向下一个创建线程的指针，以及指向挂起列表中的下一个线程的指针。图46显示了如何使用此服务。

[25]默认情况下，只有tx_thread_info_get服务可用。另外两个信息化服务必须启用才能使用它们。

```
TX_THREAD我的线程;
CHAR名称;
UINT状态;
ULONG运行_count;
UINT优先级;
UINT优先购买权_阈值;
UINT time_slep;
TX_THREAD*下一个线程;
TX_THREAD*暂停线程;
UINT状态;
…
/*检索有关以前创建的线程的信息
  “我的线程。”*/

状态=tx_thread_info_get（我的线程，名称，状态，运行计数，优先
                级，优先权，阈值，时间片，下一个线程，暂停线
                程）;

/*如果状态为TX_SUCCESS，则该信息请求有效。*/
```

图46。显示如何检索线程信息的示例

如果变量状态包含值TX_SUCCESS，则已成功检索到该信息。

优先购买权阈值变更

线程的抢占阈值可以在创建时或在运行时建立。服务tx_thread_preemption_change会更改现有线程的抢占阈值。抢占阈值可以防止优先级等于或小于优先抢占阈值的其他线程抢占线程。图47显示了如何更改抢占阈值，从而禁止任何其他线程的抢占。

在本例中，抢占阈值将更改为零(0)。这是可能的最高优先级，因此这意味着没有其他线程可以抢占此线程。但是，这并不阻止中断抢占此线程。如果我的_tlead在调用此服务之前使用了时间切片，那么该特性将被禁用。

优先级变更

创建线程时，必须为它分配一个优先级。但是，通过使用此服务，可以随时更改线程的优先级。图48显示了如何将线程my_thead的优先级更改为一个(1)。

当调用此服务时，指定线程的抢占阈值将自动设置为新的优先级。如果需要一个新的抢占阈值，则必须在优先级更改服务完成后调用tx_thread_preemption_change服务。

```
UINT我_old_阈值;
UINT状态;

/*禁用指定线程的所有抢占。这个
  当前的抢占阈值在“my_old_阈值”中返回。假设
  “my_thaed”已经创建。*/

状态=tx_thread_preemption_change(&my_thaed,
                                 0，我的旧阈值);

/*如果状态等于TX_SUCCESS，则应用程序线程不会被其他线程
  抢占。请注意，优先购买权的禁用并不会阻止isr。*/
```

图47。更改线程my_thaed的抢占阈值

```
TX_THREAD我的线程;
UINT我_old_优先级;
UINT状态;

…

/*将名为“my_shaed”的线程更改为优先级1。*/

状态=tx_thread_priority_change(&my_thaed,
                       我的老优先级);

/*如果状态等于TX_SUCCESS，则应用程序线程现在处于系统中
  的第二高优先级级别。*/
```

图48。*更改线程的优先级*

放弃控制

一个线程可以通过使用tx_thread_relinquish服务自愿将控制权移交给另一个线程。*采取此操作通常是为了实现循环调度的一种形式*。此操作是当前正在执行的线程所做的协作调用，它暂时放弃对处理器的控制，从而允许执行具有相同或更高优先级的其他线程。*这种技术有时被称为协作多线程*。下面是一个示例服务调用，它说明了线程如何将控制放弃给其他线程。

```
tx_thread_relinquish() ;
```

调用此服务提供具有相同优先级（或更高）的所有其他就绪线程在tx_thread_relinquish调用者再次执行之前执行的机会。

继续线程执行

当使用TX_DONT_START选项创建一个线程时，它将处于挂起状态。当一个线程因调用tx_thread_suspend而挂起时，它也会处于挂起状态。恢复这些线程的唯一方法是当另一个线程调用tx_thread_resume服务并将它们从挂起状态中删除时。图49说明了如何恢复一个线程。

```
TX_THREAD我的线程;
UINT状态;
…
/*恢复名为“my_theand”的线程。*/

状态=tx_thread_resume（&my_thaed）;

/*如果状态等于TX_SUCCESS，应用程序线程现在准备执行*/
```

图49。显示线程恢复my_thaed的示例

线程睡眠

在某些情况下，一个线程需要挂起一段特定的时间。这是通过tx_thread_sleep服务实现的，它将导致调用线程到

挂起为指定数量的计时器标记。下面是一个示例的服务调用
说明了一个线程如何暂停自己100个计时器答：

状态= tx_thread_sleep（100）；

如果变量状态包含值TX_SUCCESS，则当前运行的线程已挂起（或休眠）为指定数量的计时器标记。

挂起线程执行

可以通过调用tx_thread_suspend服务来挂起指定的线程。一个线程可以挂起自己，它可以挂起另一个线程，或者它也可以被另一个线程挂起。如果一个线程以这种方式挂起，那么就必须通过调用tx_thread_resume服务来恢复它。*这种类型的悬挂被称为无条件悬挂*。请注意，还有其他形式的条件挂起，例如，线程因为等待不可用的资源而被挂起，或者线程睡眠了一段特定的时间。下面是一个示例服务调用，它说明了线程（可能是它本身）如何挂起一个名为some_shead的线程。

状态=tx_thread_suspend（&some_slaed）；

如果变量状态包含值TX_SUCCESS，则指定的线程将被无条件地挂起。如果指定的螺纹已经有条件地暂停，则无条件暂停在内部保持，直到之前的暂停被解除。当解除先前的悬挂时，然后执行指定螺纹的无条件悬挂。如果指定的线程已无条件挂起，则此服务调用无效。

终止应用程序线程

此服务将终止指定的应用程序线程，而不管该线程当前是否已挂起。一个线程可以自行终止。无法再次执行已终止的线程。如果需要执行一个已终止的线程，则可以进行重置[26]

[26] tx_thread_reset服务可用于重置一个线程，以便在线程创建时定义的入口点上执行。线程必须处于已完成状态或已终止状态，才能进行重置。

或者删除它，然后再次创建它。下面是一个示例服务调用，它说明了线程（可能是它本身）如何终止线程some_shead。

状态=tx_thread_suspend（&some_slaed）；

如果变量状态包含值TX_SUCCESS，则指定的线程已被终止。

时间切片变化

线程的可选时间片可以在创建线程时进行指定，并且可以在执行期间的任何时候进行更改。此服务允许一个线程更改其自己或另一个线程的时间片。图50显示了如何创建一个时间切片
变化的

```
TX_THREAD我的线程;
ULONG我_old_time_sleck
UINT状态;
…
/*将线程“my_thoed”的时间片更改为20。*/

状态=tx_thread_time_slice_change（&我的线程，20，&我的旧
                                    _time_片）；

/*如果状态等于TX_SUCCESS，则线程时间片已更改为20，并且
  之前的时间片存储在“my_old_time_/*“*/”中
```

图50。显示线程my_thaed的时间片更改的示例

为一个线程选择一个时间片意味着在其他优先级的线程有机会执行之前，它执行的时间不会超过指定数量的计时器。请注意，如果已经指定了一个抢占阈值，则针对该线程的时间限制将被禁用。

中止螺纹悬架

在某些情况下，线程可能被迫等待不可接受的长时间（甚至永远！）对于一些资源。等待中止暂停服务可以帮助开发人员防止这种不必要的情况。此服务将中止指定线程的休眠状态或任何与等待相关的挂起。如果等待被成功中止，则a

从线程正在等待的服务中返回TX_WAIT_ABORTED值。请注意，此服务不会释放由tx_thread_suspend服务进行的显式暂停。下面是一个示例，说明了如何使用此服务：

状态=tx_thread_wait_abort（&some_slaed）；

如果变量状态包含值TX_SUCCESS，则线程some_线程的睡眠或暂停条件已中止，并且挂起的线程的返回值为TX_WAIT_ABORTED。之前挂起的线程可以自由地采取它认为适当的操作。

线程通知服务

有两个线程通知服务：tx_thread_entry_exit_notify服务和tx_thread_stack_error_notify服务。每个这些服务都注册了一个通知回调函数。如果已为特定线程成功调用了tx_thread_entry_exit_notify服务，则每次输入或退出该线程时，都会调用相应的回调函数。此功能的处理是应用程序的责任。以类似的方式，如果已经为一个特定的线程成功地调用了tx_thread_stack_error_notify服务，那么如果发生堆栈错误，将调用相应的回调函数。此错误的处理过程由应用程序负责。

执行概述

在一个Azure RTOS ThreadX应用程序中有四种类型的程序执行：

1. 设定初值
2. 线程执行
3. 中断服务例程（ISR）
4. 应用程序计时器

图51显示了每种类型的程序执行。

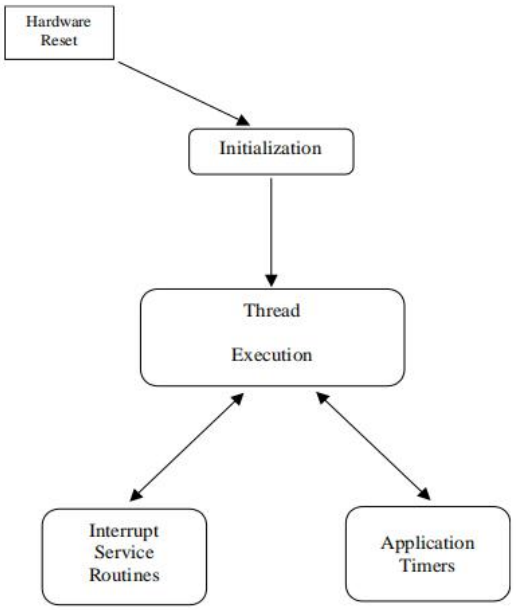


图51。程序执行类型

初始化是程序执行的第一种类型。初始化包括在处理器重置和线程调度循环的入口点之间的所有程序执行。

初始化完成后，Azure RTOS ThreadX将进入它的线程调度循环。调度循环查找一个准备执行的应用程序线程。当找到一个就绪的线程时，Azure RTOS ThreadX将控制转移给它。一旦线程完成（或另一个更高优先级的线程准备就绪），执行将转移回线程调度循环，以便找到次高优先级的已就绪线程。这个持续执行和调度线程的过程是Azure RTOS ThreadX应用程序中最常见的程序执行类型。

中断是实时系统的基石。如果不中断，很难及时对外部世界的变化作出反应。当一个中断发生时会发生什么？在检测到中断时，处理器保存有关当前程序执行的关键信息（通常在堆栈上），然后将控制转移到预定义的程序区域。这个预定义的程序区域通常被称为中断服务例程。在大多数情况下，在线程执行期间会发生中断

线程调度循环)。然而，中断也可能发生在正在执行的ISR或应用程序计时器内。

应用程序计时器与isr非常相似，除了实际的硬件实现（通常使用一个周期性的硬件中断）被隐藏在应用程序中。这些计时器被应用程序用来执行超时、定期操作和/或监督服务。就像isr一样，应用程序计时器最经常中断线程的执行。然而，与isr不同的是，应用程序计时器不能相互中断。

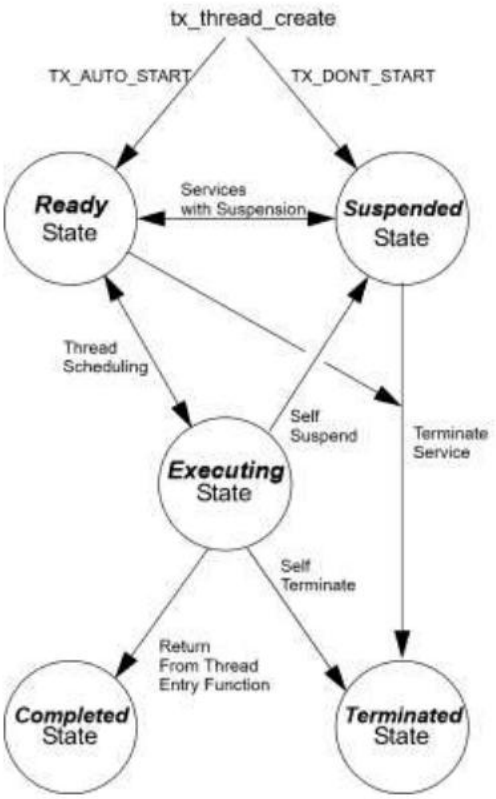


图52。线程状态转换

线程状态

理解线程的不同处理状态对于理解整个多线程环境至关重要。*有五种不同的线程状态：准备就绪，挂起，*

正在执行、*终止和完成*。图52显示了Azure RTOS ThreadX的线程状态转换图。

线程准备执行时处于就绪状态。就绪线程只有达到最高优先级的就绪线程时才会执行该线程。*当这种情况发生时，Azure RTOS ThreadX执行线程，并将其状态更改为执行*。如果高优先级线程准备就绪，执行线程将恢复到就绪状态。*然后执行新准备好的高优先级线程，这将将其逻辑状态更改为正在执行*。每次发生线程抢占时，都会发生准备状态和执行状态之间的转换。

注意，在任何给定的时刻，只有一个线程处于执行状态。这是因为处于执行状态的线程实际上可以控制底层处理器。处于挂起状态的线程不适合执行。处于挂起状态的原因包括暂停一个预定的时间；等待消息队列、信号量、互作项、事件标志或内存；以及显式线程挂起。一旦删除了暂停的原因，线程将返回到就绪状态。

如果线程处于已完成状态，这意味着线程已完成处理并从其入口函数返回。请记住，输入函数是在线程创建期间指定的。*处于已完成状态的线程不能再次执行，除非tx_thread_reset服务将其重置为其原始状态*。

一个线程处于终止状态，因为另一个线程调用tx_thread_terminate服务，或者它调用该服务本身。处于终止状态的线程不能再次执行，除非tx_thread_reset服务将其重置为其原始状态。

如第3章所述，处于就绪状态的线程有资格执行。当调度程序需要调度一个线程以进行执行时，它会选择具有最高优先级且等待时间最长的已就绪线程。如果执行的线程因任何原因中断，它的上下文将被保存，并将其置于就绪状态，准备恢复执行。驻留在挂起状态上的线程没有资格执行，因为它们正在等待不可用的资源，它们处于“睡眠”模式，它们使用TX_DONT_START选项创建，或者它们被显式挂起。

当一个挂起的线程被删除其条件时，它就有资格执行，并被移动到就绪状态。

螺纹设计

Azure RTOS ThreadX对可创建的线程的数量或可使用的优先级的组合都没有施加任何限制。但是，为了优化性能和最小化目标规模，开发人员应该遵守以下指导方针：

最小化应用程序系统中的线程数量。

仔细选择优先级。

尽量减少优先级的数量。

考虑抢占阈值。

在使用互变量时，请考虑优先级继承。

考虑循环调度。

沙痂考虑时间切片。

还有其他的指导方针，比如确保一个线程被用于完成一个特定的工作单元，而不是一系列不同的操作。

尽量减少线程的数量

一般来说，应用程序中的线程数会显著影响系统开销。这是由几个因素造成的，包括维护线程所需的系统资源数量，以及调度程序激活下一个准备线程所需的时间。每个线程，无论是否必要，都会消耗堆栈空间以及线程本身的内存空间，以及TCB的内存。

仔细选择优先级

选择线程优先级是多线程中最重要的方面之一。一个常见的错误是根据感知到的线程重要性概念来分配优先级，而不是确定在运行时实际需要什么。滥用线程优先级可能会饿死其他线程，创建优先级反转，减少处理带宽，

并使应用程序的运行时行为难以理解。如果线程饥饿是一个问题，那么应用程序可以使用添加的逻辑，逐步提高饥饿线程的优先级，直到它们有机会执行为止。然而，首先正确地选择优先级可能会显著减少这个问题。

尽量减少优先级的数量

如前所述，Azure RTOS ThreadX提供了1,024个不同的优先级值，可以分配给线程，其中默认的优先级数为32。但是，开发人员应该仔细地分配优先级，并应该根据相关线程的相对重要性来确定优先级。具有许多不同线程优先级的应用程序本身比具有较少优先级的应用程序需要更多的系统开销。回想一下，Azure RTOS ThreadX提供了一种基于优先级的优先调度算法。这意味着低优先级的线程不会执行，直到没有高优先级的线程准备执行。如果一个高优先级的线程总是准备好的，那么低优先级的线程就永远不会执行。

要理解线程优先级对上下文切换开销的影响，请考虑一个包含名为thread_1、thread_2和thread_3的线程的三线程场景。此外，假设所有线程都已挂起并等待一条消息。当thread_1收到一条消息时，它会立即将其转发给thread_2。*然后，Thread_2会将该消息转发给thread_3。Thread_3只是简单地丢弃了这个信息。*在每个线程处理其消息之后，它将再次挂起自己并等待另一条消息。执行这三个线程所需的处理根据它们的优先级而有很大的不同。如果所有线程都具有相同的优先级，则在每个线程的执行之间会发生一个上下文切换。当每个线程都挂起在一个空的消息队列上时，就会发生上下文切换。

*但是，如果thread_2的优先级高于thread_1，而thread_3的优先级高于thread_2，那么上下文切换的数量将加倍。*这是因为当tx_queue_send服务检测到一个高优先级线程已经准备好时，会发生另一个上下文切换。

如果这些线程需要不同的优先级，那么Azure RTOS ThreadX抢占阈值机制可以防止这些额外的上下文切换。这是一个重要的特性，因为它允许在线程期间有几个不同的线程优先级

调度，同时消除了在线程执行期间发生的一些不必要的上下文切换。

考虑抢占阈值

回想一下，与线程优先级相关的一个潜在问题是优先级反转。当由于低优先级线程拥有高优先级线程所需的资源而导致高优先级线程挂起时，就会发生优先级反转。在某些情况下，两个具有不同优先级的线程需要共享一个公共资源。如果这些线程是唯一活动的线程，则优先级反转时间受低优先级线程持有资源的时间的限制。这个条件既是确定性的，也是相当正常的。但是，如果一个中间优先级的一个或多个线程在这个优先级反转条件下变得活动——从而抢占低优先级线程——优先级反转时间将变得不确定性，应用程序可能会失败。

在Azure RTOS ThreadX中，有三种防止优先级反转的主要方法。首先，开发人员可以选择应用程序的优先级，并以一种防止优先级反转问题的方式设计运行时行为。其次，低优先级线程可以在与高优先级阈值线程共享资源时，利用优先优先级阈值来阻止来自中间线程的抢占。最后，使用Azure RTOS ThreadX互斥对象来保护系统资源的线程可以利用可选的互斥优先级继承来消除不确定性优先级反转。

考虑优先级继承

在优先级继承中，低优先级线程暂时获得高优先级线程的优先级，该线程试图获得低优先级线程拥有的相同互斥锁。当低优先级的线程释放互斥锁时，然后恢复其原来的优先级，并高优先级的线程获得互斥锁的所有权。该特性通过将反转时间限制到较低优先级线程保持互斥锁的时间来消除优先级反转。请注意，优先级继承只能对互斥锁可用，而对计数信号量不可用。

考虑循环调度

Azure RTOS ThreadX支持对具有相同优先级的多个线程的循环调度。这是通过对tx_thread_relinquish服务的协作调用来实现的。调用此服务将为所有其他具有相同优先级的已就绪线程提供在tx_thread_relinquish服务的调用者再次执行之前执行的机会。

考虑时间切片

时间切片为具有相同优先级的线程提供了另一种形式的循环调度。Azure RTOS ThreadX在每个线程的基础上提供时间切片。应用程序在创建线程时分配线程的时间片，并可以在运行时修改时间片。当线程的时间片过期时，所有其他具有相同优先级的准备线程都有机会在时间片线程再次执行之前执行。

螺纹内构件

当TX_THREAD数据类型用于声明一个线程时，将创建一个TCB，并将该TCB添加到一个双链接的循环列表中，如图53所示。名为tx_thread_created_ptr的指针指向列表中的第一个TCB。有关线程属性、值和其他指针，请参见TCB中的字段。

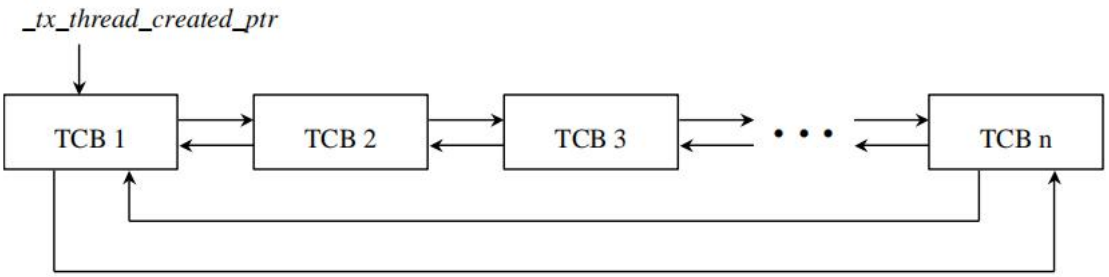


图53。已创建线程列表

当调度程序需要选择要执行的线程时，它会选择具有最高优先级的就绪线程。为了确定这样的线程，调度程序首先确定下一个已就绪线程的优先级。为了获得这个优先级，调度程序会查阅已就绪线程优先级的映射。图54显示了此优先级映射的一个示例。

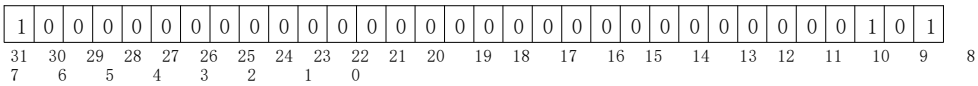


图54。显示已就绪的线程优先级的映射

这个优先级映射（实际上是一个ULONG变量）用一个比特表示32个优先级中的每一个。前面示例中的位模式表明有优先级为0、2和31的就绪线程。

当确定了下一个准备线程的最高优先级时，调度程序将使用一个准备线程头指针数组来选择要执行的线程。图55说明了这个数组的组织结构。

这个TX_THREAD列表头指针数组直接由线程优先级直接索引。如果一个条目为非空，则在该优先级上至少有一个线程准备执行。每个优先级列表中的线程都在一个双链接的循环tcb列表中进行管理，如图55所示。列表前面的线程表示要为该优先级执行的下一个线程。

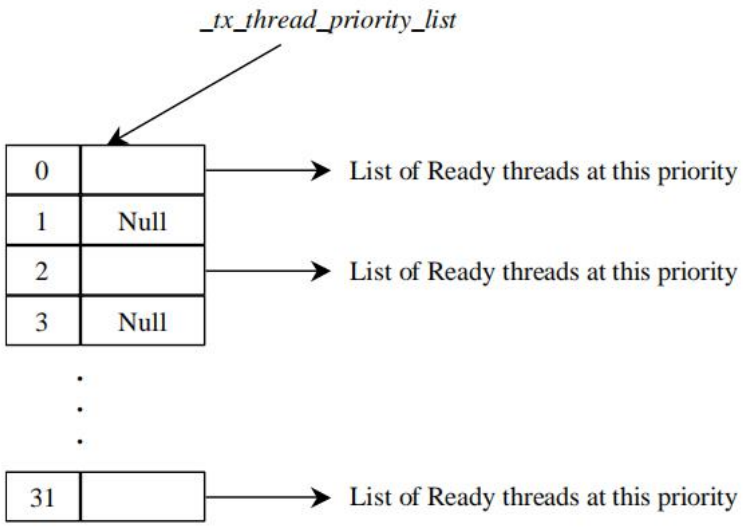


Figure 55. Example of array of ready thread head pointers indexed by priority

概述

线程是构成RTOS应用程序开发基础的动态实体。

Azure RTOS ThreadX提供了13项服务，它们被设计用于一系列的操作，包括线程的创建、删除、修改和终止。

线程控制块（TCB）是一种用于存储在执行期间有关线程状态的重要信息的结构。当上下文切换时，它还保存重要的线程信息。

应用程序开发人员在创建线程时有几个可用的选项，包括使用时间切片和抢占阈值。但是，这些选项可以在线程执行过程中随时进行更改。

一个线程可以自愿放弃控制给另一个线程，它可以恢复对另一个线程的执行，并且它可以中止对另一个线程的挂起。

一个线程有五种状态：准备就绪、挂起、执行、终止和完成。在任何时间点只执行一个线程，但可能有许多线程处于就绪状态和挂起状态。前一个列表中的线程有资格执行，而后一个列表中的线程不能执行。

主要术语和短语

中止螺纹悬挂
应用程序计时器
自动启动选项
更改抢占-阈值更改优先级
改变时间切片
已完成状态
协作的多线程优先级
就绪状态
准备好的螺纹
重入功能放弃处理器控制
创建线程
当前正在执行线程
删除线程
输入功能输入
执行状态
执行上下文
中断服务程序
优先购买权阈值
悬浮态
螺纹悬挂
终止线程服务
终止状态
螺纹控制块（TCB）

恢复线程执行	线程输入函数
循环调度	线程执行状态
服务返回值	线程运行次数
睡眠模式	线程调度循环
堆栈大小	线程启动选项
悬浮态	时间切片

问题

1. 在服务tx_thread_sleep（100）；已立即被调用之后，tx_state（TCB的一个成员）的值是多少？

2. 指定一个线程堆栈大小太小的主要危险是什么？

3. 指定一个线程堆栈大小太大的主要危险是什么？

4. 假设tx_thread_create的优先级为15，抢占阈值为20，时间片为100。这个服务呼叫的结果会是什么？

5. 给出一个例子，说明使用重入函数在多线程应用程序中是如何有用的。

6. 回答这个线程创建序列的以下问题，其中&my_stack是一个指向线程堆栈的有效指针：

```
TX_THREAD我的线程;
UINT状态;

状态=tx_thread_create（&我的线程，“我的线程”，我的线程条目，
                                    0x000F，和我的堆栈，
                                    2000,25,25,150，TX_DONT_START
                                    ）；
```

   a. 在什么时间点，这个线程将被放置在就绪状态？

   b. 是否使用优先购买权阈值？如果是这样，它的价值是什么？

   c. 是否使用时间切片？如果是这样，它的价值是什么？

   d. 线程堆栈的大小是多少？

   e. 执行此服务后，变量状态的值是多少？

7. 线程删除和线程终止之间的区别是什么？

8. 给定一个指向任意线程的指针，哪个线程服务获得该线程的状态和该线程被执行的次数？

9. 如果一个正在执行的线程的优先级为15，抢占阈值为0，那么另一个优先级为5的线程能够抢占它吗？

10. 解释时间切片和协作多线程之间的区别。

11. 在什么情况下，一个线程将被放置在就绪状态？在什么情况下，它会被移除？

12. 在什么情况下，线程会被放置在暂停状态？在什么情况下，它会被移除？

13. 请给出一个必须使用tx_thread_wait_abort服务的例子。

14. 线程调度循环如何选择要执行的线程？

15. 在什么情况下，一个线程将会被抢占？当那个线程被抢占时，它会发生什么？

16. 描述一个线程的五种状态。在什么情况下，线程的状态将从执行状态更改为就绪状态？从准备暂停？从暂停到准备？从完成到准备？

第六章

相互排斥的挑战和注意事项

介绍

在许多情况下，我们需要保证线程对共享资源或关键部分具有独占的访问权限。但是，可能需要几个线程来获得这些项目，因此我们需要同步它们的行为，以确保可以提供独占访问。在本章中，我们将考虑互斥锁的属性，它仅被设计为通过避免线程之间的冲突和防止线程之间不必要的交互来提供互斥保护。

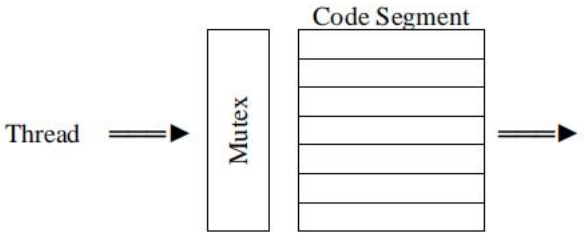
互斥锁是一种公共资源，最多可以由一个线程拥有。此外，一个线程（只有同一线程）可以重复[27]获得相同的互斥锁2^{32}-准确地说，这是1倍。但是，相同的线程（并且只有那个线程）必须放弃在互斥锁再次可用之前的相同次数的互斥锁。

保护关键部分

关键部分是一个代码段，其中指令必须按顺序执行而不中断。互斥锁有助于实现这个目标。考虑图56，它显示了一个作为关键部分的代码段。要进入此临界部分，线程必须首先获得保护临界部分的某个互斥锁的所有权。因此，当线程准备好开始执行这个代码段时，它首先尝试获取它

[27] 一些作者将这种类型的互斥体描述为递归互斥体，因为它具有同线程、多重所有权的能力。但是，我们将不会在这里使用这个术语。

互斥。在线程获得互斥锁后，它将执行代码段，然后放弃该互斥锁。



n

提供对共享资源的独占访问权限

互斥锁可以以保护关键部分的方式提供对共享资源的独占访问。也就是说，线程必须先获得互斥锁，然后才能访问共享资源。但是，如果一个线程必须同时具有对两个（或多个）共享资源的独占访问权限，那么它必须使用一个单独的互斥锁来保护每个共享资源。在这种情况下，线程必须首先为每个共享资源获得一个特定的互斥锁，然后才能继续。图57说明了这个过程。当线程准备好访问这些资源时，它首先获得保护这些资源的两个互变量。在线程获得了这两个互变量之后，它将访问共享的资源，然后在完成这些资源后放弃这两个互变量。

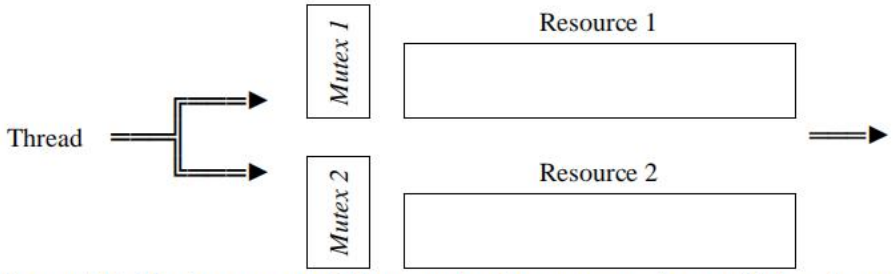


Figure 57. Mutexes providing exclusive access to multiple shared resources

互斥控制块

互斥锁控制块（MCB）[28]是一种用于在运行时维护互斥锁状态的结构。它包含多种信息，包括互斥锁所有者、所有权计数、优先级继承标志、拥有线程的原始优先级、拥有线程的原始抢占阈值、挂起计数和指向挂起列表的指针。图58包含了组成MCB的许多字段。

场地	描述
tx_mutex_id	控制块ID
tx_mutex_name	指向互斥名称的指针
tx_mutex_ownership_count	Mutex所有权数量
* tx_mutex_owner	Mutex所有权指针
tx_mutex_inherit	优先级继承标志
tx_mutex_original_priority	拥有线程的原始优先级
tx_mutex_original_threshold	拥有线程的原始阈值
* tx_mutex_suspension_list	悬挂列表中的指针
tx_mutex_suspended_count	暂停列表计数
* tx_mutex_created_next	指向已创建列表中的下一个互斥锁的指针
* tx_mutex_created_previous	指向已创建列表中的前一个互斥锁的指针

图58。互斥控制块

在大多数情况下，开发人员可以忽略MCB的内容。但是，在某些情况下，特别是在调试期间，检查MCB的某些成员是很有用的。请注意，尽管Azure RTOS ThreadX允许检查MCB，但它严格禁止修改MCB。

互斥服务概要

附录E包含了关于互斥服务的详细信息，提供了以下信息：原型、服务的简要描述、参数、返回值、注释和警告、允许调用、抢占可能性和an

[28]每个互斥体的特征都包含在其MCB中。这个结构在tx_api中定义了。h文件。

说明如何使用该服务的示例。图59包含了所有可用的互斥锁服务的列表。在本章的以下部分中，您将研究这些服务。我们将考虑这些服务的许多特性，并将开发一个使用它们的示例系统的示例。

互斥服务	描述
tx_mutex_create	创建一个互斥
tx_mutex_delete	删除一个互斥
tx_mutex_get	试图获得互斥锁的所有权
tx_mutex_info_get	检索有关互斥锁的信息
tx_mutex_prioritize	将最高优先级的悬挂螺纹放在悬挂列表的前面
tx_mutex_put	发布互斥的所有权

图59。互斥服务

创建一个互斥

互斥体用数据类型TX_MUTEX声明[29]，并由tx_mutex_create服务进行定义。在定义互斥锁时，您需要指定MCB、互斥锁的名称和优先级继承选项。

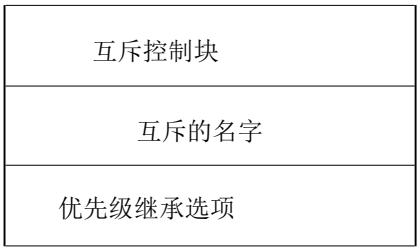


图60。互斥的属性

图60包含了这些属性的列表。我们将开发一个互斥体创建的例子来说明此服务的使用。我们将把互斥项命名为“my_mu互斥项”，并将激活优先级继承特性。优先级继承允许低优先级线程暂时承担高优先级线程的优先级，该线程正在等待低优先级线程拥有的互斥锁。该特性通过消除中间程序的抢占，帮助应用程序避免优先级反转

[29] 当声明一个互斥体时，将创建一个MCB。

线程优先级。图61包含了一个具有优先级继承的互斥锁创建的示例。

如果您想创建一个没有优先级继承特性的互斥体，那么您将使用TX_NO_INHERIT参数，而不是TX_INHERIT参数。

```
TX_MUTEX我_mutex;
UINT状态;

/*创建一个互斥锁，以提供对共享资源的保护。*/

状态=tx_mutex_create（&我的_mutex，“我的_mutex_name”
                          ，TX_INHERIT）；

/*如果状态等于TX_SUCCESS，则my_mutex就可以使用*/了
```

图61。创建具有优先级继承的互斥锁

删除一个互斥

可以使用tx_mutex_delete服务删除互斥锁。当一个互斥锁被删除时，所有因为它们正在等待该互斥锁而被挂起的线程都会恢复（即，处于就绪状态）。每个这些线程都将从其对tx_mutex_get的调用中接收到一个TX_DELETED返回状态。图62包含一个说明如何删除名为“my_mutex”的互斥例。

```
TX_MUTEX我_mutex;
UINT状态;
…
/*删除互斥。假设互斥锁已经存在了
  创造的*/

状态= tx_mutex_delete（&my_mutex）；

/*如果状态为TX_SUCCESS，则该互斥锁已被删除。*/
```

图62。删除互斥

获得互斥锁的所有权

tx_mutex_get服务允许线程尝试获得互斥锁的独占所有权。如果没有线程拥有该互斥锁，那么该线程将获得该互斥锁的所有权。

如果调用线程已经拥有互斥锁，则tx_mutex_get将增加所有权计数器并返回成功状态。如果另一个线程已经拥有该互斥机，则所采取的操作取决于与tx_mutex_get一起使用的调用选项，以及该互斥机是否启用了优先级继承。这些操作如图63所示。

tx_mutex_get 等待选项	在互斥锁中启用了 优先级继承	在互斥锁中禁用优 先级继承
tx_no_wait	立即返回	立即返回
tx_wait_forever	如果拥有线程具有更高的优先级，则将调用线程的优先级提升到拥有线程的优先级，则调用线程被放置在挂起列表中，调用线程无限期地等待	线程放置在 暂停列表和 无限期等待
超时值	如果拥有线程具有更高的优先级，则将调用线程的优先级提升到拥有线程的优先级，然后将调用线程放在挂起列表中，并且调用线程等待，直到指定的计时器标记数过期	放置在挂起列表上的线程，并等待指定的计时器的数量过期

图63。当互斥锁已被另一个线程拥有时所采取的操作

如果使用优先级继承，请确保不允许其他线程修改在互斥锁所有权期间继承了更高优先级的线程的优先级。图64包含了一个试图获得互斥锁所有权的线程的示例。

如果变量状态包含值TX_SUCCESS，则这是一个成功的获取操作。在本例中使用了TX_WAIT_FOREVER选项。因此，如果互斥锁已经被另一个线程拥有，那么调用线程将在挂起列表中无限期地等待。

```
TX_MUTEX我_mutex;
UINT状态;
…
/*获得互斥器“my_互斥器”的独家所有权。如果tex调用
  “my_mutex”不可用，直到可用。*/

状态= tx_mutex_get（&my_mutex，TX_WAIT_FOREVER）;
```

图64。获得互斥锁的所有权

检索互斥信息

有三种服务允许您检索有关互斥关系的重要信息。第一个这样的互斥体服务——tx_mutex_info_get服务——从互斥体控制块中检索信息的子集。此信息在特定时刻，即调用服务时提供“快照”。另外两个服务提供了基于运行时性能数据的收集的汇总信息。其中一个服务——tx_mutex_performance_info_get服务——在调用服务之前为特定的互斥锁提供信息摘要。相比之下，tx_mutex_performance_system_info_get会检索到调用服务之前系统中所有互变量的信息摘要。这些服务对于分析系统的行为和确定是否存在潜在的问题领域很有用。tx_mutex_info_get[30]服务获取的信息包括所有权计数、拥有线程的位置、第一个线程在暂停列表中的位置、挂起线程的数量以及下一个创建的互斥锁的位置。图65显示了如何使用此服务的关于互斥锁的当前信息。

```
TX_MUTEX我_mutex;
CHAR名称;
ULONG计数;
TX_THREAD*所有者;
TX_THREAD*first_暂停;
ULONG暂停_count;
TX_MUTEX*下一个_mutex;
UINT状态;
…
/*检索关于之前创建的“my_互斥体”的信息。*/

状态=tx_mutex_info_get（&我的互斥系统，姓名，计数，所有者，首先暂停，暂停计数
                          ，和下一个互斥系统）；

/*如果状态为TX_SUCCESS，则信息已成功接收到*/
```

图65。显示如何检索互斥锁信息的示例

[30]默认情况下，只有tx_mutex_info_get服务可用。另外两个信息化服务必须启用才能使用它们。

如果变量状态包含值TX_SUCCESS，则已成功检索到该信息。

优先考虑互斥锁暂停列表

当线程因为等待互斥锁而挂起时，将以FIFO方式将其放置在适当的挂起列表中。当互斥锁可用时，挂起列表中的第一个线程（不论优先级）将获得该互斥锁的所有权。tx_mutex_prioritize服务将最高优先级的线程放置在挂起列表的前面。所有其他线程保持挂起时的FIFO顺序。图66显示了如何使用此服务。

```
TX_MUTEX我_mutex;
UINT状态;
…
/*确保最高优先级的线程将获得该文件的所有权
当互斥体可用时。*/

状态= tx_mutex_prioritize（&my_mutex）；

/*如果状态为TX_SUCCESS，则最高优先级的挂起线程已放在列表的前面。下
  一个释放互斥锁所有权的tx_mutex_put调用将赋予这个线程所有权，并将
  其唤醒为*/
```

图66。确定互斥锁暂停列表的优先级

如果变量状态包含值TX_SUCCESS，则挂起列表中等待互斥锁“my_mutex”的最高优先级线程已被放置在挂起列表的前面。如果没有线程等待这个互斥锁，返回值也是TX_SUCCESS，挂起列表保持不变。请注意，此服务不会对挂起列表进行排序——它只是将最高优先级的线程移动到列表的前面。

释放一个互斥锁的所有权

tx_mutex_put服务允许线程释放互斥锁的所有权。假设线程拥有互斥锁，那么所有权计数就会减少。如果所有权计数变为零，则互斥锁将成为可用的。如果互斥锁可用，如果

为这个互斥锁启用了优先级继承，然后释放线程的优先级恢复到它最初获得互斥锁所有权时的优先级。在互斥锁所有权期间对释放线程所做的任何其他优先级更改也可以撤消。图67显示了如何使用此服务。

```
TX_MUTEX我_mutex;
UINT状态;
…
/*释放了“my_mutex”的所有权。*/

状态= tx_mutex_put（&my_mutex）;

/*如果状态等于TX_SUCCESS，则互斥锁所有权计数已减少
  ，如果为零，则已释放。*/
```

图67。释放互斥锁的所有权

如果变量状态包含值TX_SUCCESS，则put操作成功，所有权计数将减少。

避免致命的拥抱

使用互惠音的潜在缺陷之一[31] *也就是所谓的那种致命的拥抱*。这是一种不受欢迎的情况，即在试图获得其他线程已经拥有的互斥体时，两个或更多个线程会被无限期地挂起。图68说明了一个导致致命拥抱的场景。以下是本图中所描述的事件序列。

1. 线程1获得了互斥体1的所有权
2. 线程2获得了互斥体2的所有权
3. 线程1暂停是因为它试图获得Mutex 2的所有权
4. 线程2暂停是因为它试图获得Mutex 1的所有权

因此，线程1和线程2进入了一个致命的拥抱，因为它们无限期地暂停，每个都等待另一个线程拥有的互斥锁。

[31] 这个问题也与信号量的使用有关，我们将在后面的章节中讨论它。

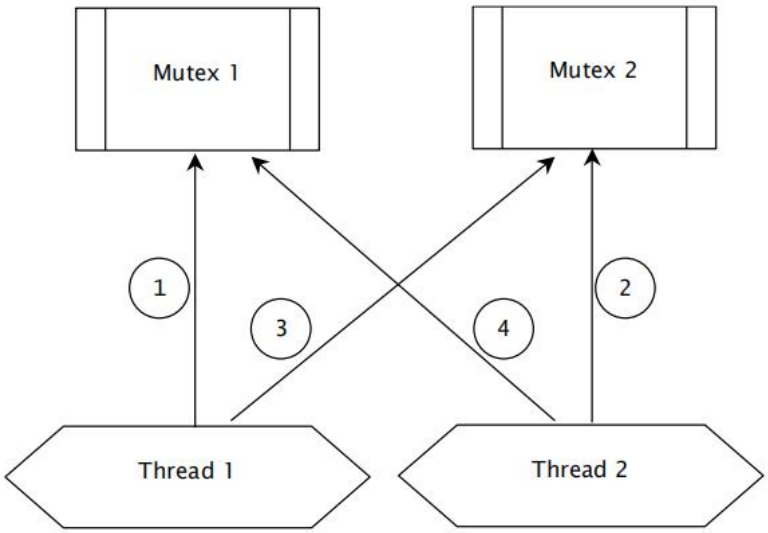


图68。导致致命拥抱的一系列行动

你如何才能避免致命的拥抱呢？在应用程序级别上的预防是实时系统的唯一方法。保证没有致命拥抱的唯一方法是允许一条线在任何时候最多拥有一个互斥体。如果线程必须拥有多个互调，如果使线程以相同的顺序收集互调，通常可以避免致命的拥抱。例如，如果线程总是以连续的顺序获得两个互斥体，即线程1（或Thread 2）将尝试获取Mutex 1，然后立即尝试获取Mutex 2，则可以阻止图68中的致命拥抱。另一个线程将尝试以相同的顺序获取Mutex 1和Mutex 2。在任何一种情况下，一个线程都不会保持一个互斥锁的所有权，除非它也可以获得另一个互斥锁的所有权。

从致命的拥抱中恢复的一种方法是使用与tx_mutex_get服务相关的暂停超时功能，这是三个可用的等待选项之一。另一种从一个致命的拥抱中恢复过来的方法是让另一个线程调用tx_thread_wait_abort服务来中止一个被困在一个致命的拥抱中的线程的挂起。

使用互斥体来保护临界部分的样本系统

我们将创建一个示例系统来说明如何使用互斥锁来保护两个线程的临界部分。这个系统在第二章中介绍，快速线程和慢线程各有四个活动，其中两个是关键的

部分图69和图70显示了这两个线程的活动序列，其中阴影框表示关键部分。

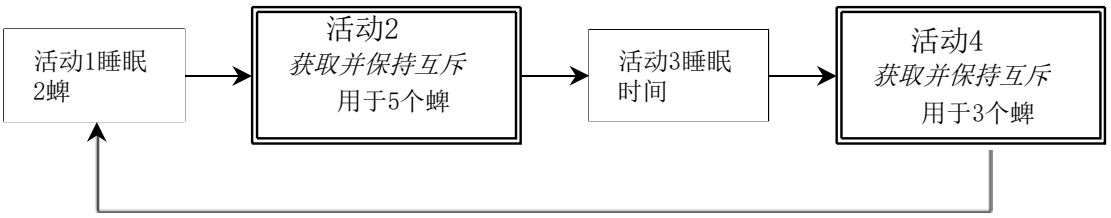


图69。快速线程的活动（优先级= 5）

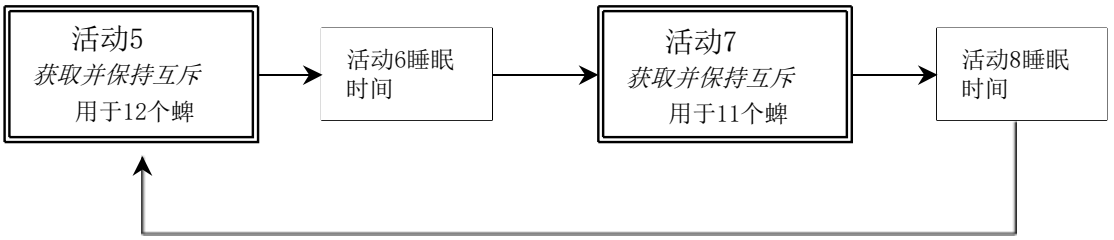


图70。慢线程的活动（优先级= 15）

图71。基本系统结构

为了开发这个系统，我们将需要创建两个线程和一个互斥锁。每个线程必须有自己的堆栈，我们将将其实现为一个数组，而不是一个内存字节池。我们将需要创建可执行所需活动的线程输入函数。因为我们将完整地创建这个系统，所以我们用图71概述了这个过程，这是在第2章中首次出现的基本四部分系统结构的变体。

对于系统的第一部分，我们将两个线程、一个互斥锁和两个线程堆栈声明为全局实体，如下所示：

```
TX_THREAD快速线程，慢线程；
TX_MUTEX我_mutex；
#定义STACK_SIZE 1024；
CHARstack_快速[STACK_SIZE]，stack_slow[STACK_SIZE]；
```

声明线程的过程创建了两个线程控制块（tcb），并且声明互斥锁也创建了它的MCB。线程堆栈将准备好在tx_application_define函数中使用。

系统的第二部分是我们定义主入口点的地方，它是对进入Azure RTOS ThreadX内核的调用。

系统的第三部分是我们定义线程和互斥锁的地方。下面是spephy_thead的定义。

```
tx_thread_create(和“快速线”，
                      Speedy_Thread_条目，0
                      堆栈速度，STACK_SIZE，
                      5，5，tx_no_time_slice ，tx_auto_start )；
```

Speedy_线程的优先级为5，但没有抢占阈值，也没有时间片。下面是slow_sload的定义。

```
tx_thread_create(&“慢线”，“慢线”，
                      Slow_Thread_入口，1，
                      stack_slow STACK_SIZE
                      15，15，tx_no_time_slice ，tx_auto_start )；
```

Slow_sloade的优先级为15，但没有抢占阈值，也没有时间片。这两个线程都将立即启动。下面是my_mutex的定义。

tx_mutex_create（&my_mutex，“my_mutex”，TX_NO_INHERIT）；互斥体有一个名称，但没有优先级继承特性。

我们系统的第四部分是我们开发线程输入函数的地方。下面是spepy_thoaed的入口函数的一部分。

```
/*活动1： 2计时器-滴答声。*/
    tx_thread_sleep (2);
/*活动2-关键部分-5计时器标记
    获得互斥锁与悬挂系统*/
tx_mutex_get（&my_mutex，TX_WAIT_FOREVER）；
tx_thread_sleep (5);
/*释放互斥锁*/
tx_mutex_put（&my_mutex）；
```

这里表示了快速线程的前两个活动。活动1不是一个关键的部分，所以我们立即睡眠两个计时器。活动2是一个关键的部分，所以要执行它，我们必须首先获得互斥锁的所有权。在我们得到互斥质后，我们睡了五分钟。这两个线程的其他活动都遵循类似的模式。当我们开发完整的系统时，我们将检查返回值的状态，以确保服务调用已经正确执行。

图72到图76包含了这个示例系统的完整列表，分为五个部分，其中最后两个部分是线程输入函数。完整的程序列表称为06_sample_system。c位于本章的后面一节中。

```
/* 06_sample_system .c
  创建两个线程和一个互斥锁。
  使用线程堆栈的数组。
  互斥锁保护了临界部分。*/

#include "tx_api.h"
#include <stdio.h>

#定义STACK_SIZE 1024

CHAR堆栈_speedy[STACK_SIZE];
CHAR堆栈_slow[STACK_SIZE];

/*定义Azure RTOS ThreadX对象控件
赛跑者起跑时脚底
所撑的木块*/          快速的线程;
tx_thread             慢线;
tx_thread             我的_mutex;
tx_mutex

/*定义线程原型。*/
空白快速线程输入（ULONG线程_intut）; 空白慢线程
输入（ULONG线程输入）;
```

图72。定义、声明和原型

示例系统的第一部分（图72）包含了所有必要的指令、声明、定义和原型。

示例系统的第二部分（图73）包含主要入口点。这是进入Azure RTOS ThreadX内核的条目。请注意，对tx_kernel_enter的调用不会返回，因此不要在它后面放置任何处理。

```
/*定义主要入口点。*/

int主（）
{

  /*输入Azure RTOS ThreadX内核。
*/
  tx_kernel_enter() ;
}
```

图73。主要入口点

样本系统的第三部分（图74）由称为tx_application_define的应用程序定义函数组成。此函数可以用来定义系统中的所有应用程序资源。这个函数有一个输入参数，这是第一个可用的RAM地址。这通常被用作线程堆栈、队列和内存池的初始运行时内存分配的起点。

示例系统的第四部分（图75）由spedy_thoaed的入口函数组成。此函数定义了线程的四个活动，并显示线程完成一个完整周期时的当前时间。

```
/*定义初始系统的外观。*/

空白tx_application_define（空白*未使用内存）
{
  /*将系统定义放在这里，e。g.，线程和互斥锁创建了*/

  /*创建Speedy_Thread。*/
  tx_thread_create(和“快速线”，
                   Speedy_Thread_条目，0
                   堆栈速度，STACK_SIZE，
                   5, 5, tx_no_time_slice , tx_auto_start );

  /*创建慢程*/
  tx_thread_create(&“慢线”，“慢线”，
                   Slow_Thread_入口，1，
                   stack_slow STACK_SIZE
                       15, 15, tx_no_time_slice , tx_auto_start );

  /*创建由两个线程*/tx_mutex_create所使用的互斥锁（&my_mutex
  ，“my_mutex”，TX_NO_INHERIT）；
}
```

图74。应用程序定义

```
/*为快速线程*/定义活动

空白快速线程输入（ULONG线程输入）
{
UINT状态；
ULONG电流时间；

而(1)
{
/*活动1： 2计时器-滴答声。*/
tx_thread_sleep (2);

/*活动2-关键部分-5计时器标记-获得互斥锁与暂停。*/状态= tx_mutex_get（&my
_mutex，TX_WAIT_FOREVER）；
如果（状态！= TX_SUCCESS）中断；/*检查状态*/
tx_thread_sleep (5);
/*释放互斥。*/
状态= tx_mutex_put（&my_mutex）；
如果（状态！= TX_SUCCESS）中断；/*检查状态*/

/*活动3： 4计时器-计时器。*/
tx_thread_sleep (4);

/*活动4-关键部分-3计时器标记
    获得互斥锁。*/
    状态= tx_mutex_get（&my_mutex，TX_WAIT_FOREVER）；
如果（状态！= TX_SUCCESS）中断；/*检查状态*/
tx_thread_sleep (3);
/*释放互斥。*/
状态= tx_mutex_put（&my_mutex）；
如果（状态！= TX_SUCCESS）中断；/*检查状态*/
当前时间= tx_time_get（）；
打印f（“当前时间：%luSpeedy_线程完成周期...\n”，当前时间）；}
}
```

图75。Speedy_ Thread输入功能

示例系统的第五部分也是最后一部分（图76）由slow_tlaed的输入功能组成。此函数定义了线程的四个活动，并显示线程完成一个完整周期时的当前时间。

```
/*为Slow_Thread */定义活动

空白慢线程输入（ULONG线程输入）
{
UINT状态；
ULONG电流时间；

  而(1)
  {

    /*活动5-关键部分-12计时器答-获得互斥锁与暂停。*/状态= tx_mutex_get（&my_mu
    tex，TX_WAIT_FOREVER）；
    如果（状态！= TX_SUCCESS）中断；/*检查状态*/
    tx_thread_sleep (12);
    /*释放互斥。*/
    状态= tx_mutex_put（&my_mutex）；
    如果（状态！= TX_SUCCESS）中断；/*检查状态*/

    /*活动6： 8计时器-计时器。*/
    tx_thread_sleep (8);

    /*活动7-关键部分-11个计时器标记
    获得互斥锁。*/
    状态= tx_mutex_get（&my_mutex，TX_WAIT_FOREVER）；
    如果（状态！= TX_SUCCESS）中断；/*检查状态*/
    tx_thread_sleep (11);
    /*释放互斥。*/
    状态= tx_mutex_put（&my_mutex）；
    如果（状态！= TX_SUCCESS）中断；/*检查状态*/

    /*活动8： 9计时器-计时器。*/
    tx_thread_sleep (9);
    当前时间= tx_time_get（）；
     （“当前时间：%lu慢线程完成周期...\n”，当前时间）；）
}
```

图76。Slow_ Thread输入函数

由样品系统产生的输出

图77包含了通过执行示例系统执行几个线程活动周期所产生的一些输出。您的输出应该是相似的，但不一定是完全相同的。

```
当前时间：      34Speedy_线程完成周期...
当前时间：      40慢线程完成循环...
当前时间：      56Speedy_线程完成周期...
当前时间：      77Speedy_线程完成周期...
当前时间：      83慢线程完成循环...
当前时间：      99Speedy_线程完成周期...
当前时间：      120快速程完成周期...
当前时间：      126慢线程完成循环。..
当前时间：      142快速程完成循环...
当前时间：      163快速程完成循环...
```

图77。样品系统产生的部分输出

快速线程完成其活动周期所需的最短时间是14个计时器点。相比之下，慢线程至少需要40个计时器来完成其活动的一个活动周期。但是，慢线程的关键部分会导致快速线程的延迟。考虑图77中的示例输出，其中spedy_线程在时间34结束了它的第一个周期，这意味着由于慢线程，它遇到了20个计时器的延迟。快速线程会以更及时的方式完成后续的循环，但它总是会花费大量的时间来等待慢线程来完成它的关键部分。

为了更好地理解样本系统中发生了什么，让我们跟踪所发生的一些操作。初始化完成后，两个线程都处于就绪状态，并准备执行。调度程序选择快速线程进行执行，因为它比慢线程有更高的优先级。Spedy_thaed开始活动1，这导致它睡眠两个计时器，即在此期间它被放置在暂停状态上。然后Slow_sleak开始执行，它开始了活动5，这是一个关键的部分。slow_thaed拥有互斥锁，然后睡眠12次，i。e.，在此期间，它将处于暂停状态。在时间2时，spedy_shaed从暂停状态移除，放置于就绪状态，并开始活动2，这是一个关键部分。快速线程试图获得互斥锁的所有权，但它已经被拥有，因此快速线程被放置在挂起状态，直到互斥锁可用。在时间12，慢_slead被放置回就绪状态并放弃互斥锁的所有权。图78包含了样本系统的操作的部分跟踪。

时间	执行的操作	麦克斯业主
初始	*快速的和缓慢的速度都已经准备好了*	没有一个
0	快速睡2个，暂停TSL，缓慢采取互斥体，睡12个和暂停	*慢的*
2	快速的已经准备好了，无法得到互斥锁，暂停	*慢的*
12	慢就准备好了，放弃互斥锁，快速抢占慢， 快速采取互斥质，睡5岁，暂停，慢睡 8和悬挂	*快的*
17	快速准备好，放弃互斥，睡4和暂停	没有一个
20	慢准备好，互斥，睡11和暂停	*慢的*
21	快速的已经准备好了，无法得到互斥锁，暂停	*慢的*
31	慢就准备好了，放弃互斥锁，快速抢占慢， 快速采取互斥质，睡3岁，暂停，慢睡 9和暂停	*快的*
34	快速准备好，放弃互斥，睡3个，暂停 （这就完成了Speedy的一个完整周期）	没有一个
37	快速的已经准备好了，睡了2个小时和暂停	没有一个
39	快速准备好了，采取互斥质，睡5人和暂停	*快的*
40	慢速已经准备好了，无法获得互斥锁，无法暂停 （这就完成了一个缓慢的完整周期）	*快的*

图78。样品系统的部分活动跟踪

06_sample_system清单。c

06_sample_system.c的完整清单如下；已添加了行编号，以方便参考。

```
001 /* 06_sample_system .c
002
创建两个线程和一个互斥锁。
004为线程堆栈使用一个数组。
互斥锁保护临界部分。*/
006
007 /**************************************************/
声明、定义和原型*/
009 /**************************************************/
```

```
010
011 #include "tx_api.h"
012 #include <stdio.h>
013
定义STACK_SIZE 1024
015
016CHARstack_speeedy[STACK_SIZE];
017年CHAR堆栈_slow[STACK_SIZE];
018
019
020 /*定义Azure RTOS ThreadX对象控件块*/...
021
022年TX_THREAD快速的线程;
023 TX_THREAD慢线程; 024
025 TX_MUTEX my_mutex;
026
027
定义线程原型。*/
029
030空的快速输入（ULONG线程输入）;
031空白慢线程输入（ULONG线程输入）; 032
033
034 /**************************************************/
035 /*主入口点*/
036 /**************************************************/
037
定义主要入口点。*/
039
040 int主（）
041 {
042
043 /*输入Azure RTOS ThreadX内核。*/
044 tx_kernel_enter() ;
045 }
046
047
048 /**************************************************/
```

```
应用程序定义*/
050 /**************************************************/
```

051
定义初始系统的外观。*/
053
054空白tx_application_define（空白*第一个未使用的内存）
055 {
056
057
把系统定义放在这里，
059 e.g., 线程和互斥锁创建了*/
060
创建快速的线程。*/
062 tx_thread_create(&“快速线”，“快速线”，
快速的线程条目，0，
堆栈的速度，STACK_SIZE，
065 5, 5, tx_no_time_slice , tx_auto_start);
066
创建慢线程*/
068 tx_thread_create(&“慢线”，“慢线”，
慢程，1，
070 stack_slow，STACK_SIZE，
071 15, 15, tx_no_time_slice , tx_auto_start);
072
创建两个线程*/所使用的互斥锁
074 tx_mutex_create（“&my_互斥”，“my_mu互斥””，TX_NO_INHERIT）；
075
076 }
077
078
079 /**/
080 /*函数定义*/
081 /**/
082
为快速线程*/定义活动
084
085空白快速输入（ULONG线程输入）
086 {
087年UINT状态；
088年ULONG当前_time；089
090时(1)

091 {

```
092
活动1： 2计时器。*/
094 tx_thread_sleep (2);
095
活动2： 5计时器计时***关键部分***
097获得互斥锁。*/
098
099状态= tx_mutex_get（&my_mutex，TX_WAIT_FOREVER）；
如果状态！=TX_SUCCESS)(中断；/*检查状态*/ 101
102 tx_thread_sleep (5);
103
释放互斥锁。*/
105状态= tx_mutex_put（&my_mutex）；
106如果（状态！= TX_SUCCESS)中断；/*检查状态*/ 107
活动3： 4计时器。*/
109 tx_thread_sleep (4);
110
活动4： 3计时器计时***关键部分***
112获得互斥锁。*/
113
114状态= tx_mutex_get（&my_mutex，TX_WAIT_FOREVER）；
115如果（状态！= TX_SUCCESS）中断；/*检查状态*/ 116
117 tx_thread_sleep (3);
118
释放互斥锁。*/
120状态= tx_mutex_put（&my_mutex）；
121如果（状态！= TX_SUCCESS）中断；/*检查状态*/ 122
123电流_time=tx_time_get（）；
“当前时间：%luSpeedy_线程完成周期...\n”，
125年当前_时间)；
126
127     }
128 }
129
130 /**************************************************/
131
定义慢线程*/的活动
```

```
133
134空白慢线程输入（ULONG线程输入）
135 {
136年UINT状态；
137 ULONG当前_time；138
139时(1)
140    {
141
活动5： 12计时器，关键部分***
143获得互斥锁。*/
144
145状态= tx_mutex_get（&my_mutex，TX_WAIT_FOREVER）；
146如果（状态！= TX_SUCCESS）中断；/*检查状态*/ 147
148 tx_thread_sleep (12);
149
释放互斥锁。*/
151状态= tx_mutex_put（&my_mutex）；
152如果（状态！= TX_SUCCESS）中断；/*检查状态*/ 153
活动6： 8计时器。*/
155 tx_thread_sleep (8);
156
活动7： 11计时器，关键部分***
158获得互斥锁。*/
159
160状态= tx_mutex_get（&my_mutex，TX_WAIT_FOREVER）；
161如果（状态！= TX_SUCCESS)中断；/*检查状态*/ 162
163 tx_thread_sleep (11);
164
释放互斥锁。*/
166状态= tx_mutex_put（&my_mutex）；
167如果（状态！= TX_SUCCESS）中断；/*检查状态*/ 168
活动8： 9计时器。*/
170 tx_thread_sleep (9);
171
172电流_time=tx_time_get（）；
“当前时间：%luSlow_线程完成周期...\n”，
```

```
174当前_time);
175
176     }
177 }
```

互感器内部构件

当TX_MUTEX数据类型用于声明一个互斥体时，将创建一个MCB，并将该MCB添加到一个双链接的循环列表中，如图79所示。

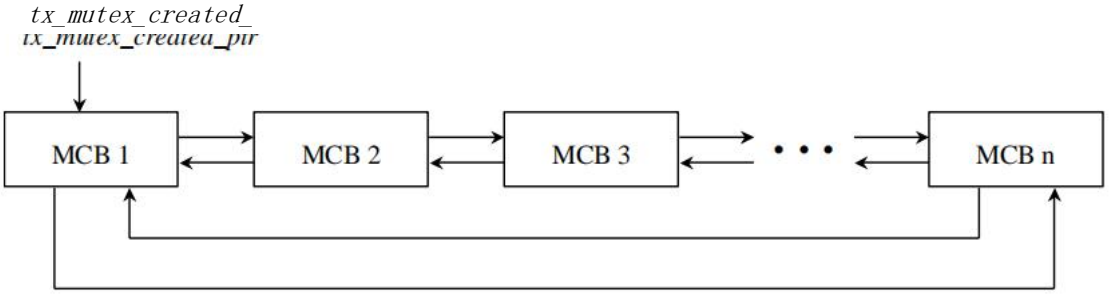


图79。已创建的互斥列表

名为tx_mutex_created_ptr的指针指向列表中的第一个MCB。有关互斥锁属性、值和其他指针，请参见MCB中的字段。

行动优先级

	我的线程
我的线程获得我的mutex	25
大线程（优先级= 10）试图获得我的互斥锁，但被挂起，因为互斥锁属于我的线程，它继承了优先级10，而它拥有my_互斥锁	10
my_thae将自己的优先级改为15	15
my_thae将自己的优先级改为21	21
my_trone释放my_mutex	25

图80。显示优先级继承对线程优先级的影响的示例

如果已指定了优先级继承特性（即，已设置了名为tx_mutex_inherit的MCB字段），则拥有线程的优先级将增加到

匹配一个挂起在这个互斥锁上的具有更高优先级的线程。当拥有线程释放互斥锁时，其优先级恢复到其原始值，而不管中间优先级更改。考虑图80，它包含了名为my_tlied的线程的一系列操作序列，它成功地获得了名为my_mutex的互斥锁，它启用了优先级继承特性。

名为my_thead线程的线程初始优先级为25，但它继承了名为big_线程的优先级10。在这一点上，我的线程两次改变了它自己的优先级（也许是不明智的，因为它降低了它继承的优先级！）。当my_tlaed释放互斥锁时，它的优先级恢复到原来的值25，尽管中间优先级发生了变化。请注意，如果my_tloed之前指定了一个抢占阈值，那么当执行更改优先级操作时，新的优先阈值将被更改为新的优先级。当my_sloed释放互斥锁时，抢占阈值将被更改为原始的优先级值，而不是原始的优先阈值。

概述

互斥锁是一种公共资源，在任何时候最多可以由一个线程拥有。它只有一个目的：提供对关键部分或共享资源的独占访问。

声明一个互斥体具有创建一个MCB的效果，该MCB是一种用于在执行期间存储关于该互斥体的重要信息的结构。

有8种服务设计用于涉及互斥的一系列操作，包括创建互斥、删除互斥、确定暂停列表的优先级、获得互斥的所有权、检索互斥信息(3)以及放弃互斥的所有权。

开发人员可以在定义互斥锁时或在以后的执行过程中指定优先级继承选项。使用此选项将减少优先级反转的问题。

与使用互音相关的另一个问题是致命的拥抱，并提出了几个避免这个问题的建议。

我们开发了一个完整的系统，它使用两个线程和一个互斥锁来保护每个线程的临界部分。我们提出并讨论了这些线程的部分跟踪。

主要术语和短语

创建互斥
临界断面
死锁
删除互斥
排斥存取
多种互斥权
麦克斯
互斥锁控制块（MCB）互斥
锁等待选项
互斥现象
互斥的所有权
优先排序互斥暂停列表
优先权继承性
优先反演
就绪状态
从致命拥抱中恢复
共享资源
悬浮态
同步线程行为

问题

1. 精确地描述以下互斥体声明所发生的结果： TX_MUTEX mutex_ 1；

2. 互斥锁声明和互斥锁创建的区别有什么区别？

3. 假设一个互斥锁不被拥有，并且一个线程与tx_mutex_get服务获取该互斥锁。tx_mutex_suspended_count（MCB的一个成员）在该服务完成后立即获得的价值是多少？

4. 假设一个具有最低可能优先级的线程拥有某个互斥锁，而一个具有最高可能优先级的已准备好的线程需要该互斥锁。高优先级线程会成功地从低优先级线程中获取该互斥锁吗？

5. 描述将导致执行线程移动到挂起状态的所有情况（迄今已讨论）。

6. 假设一个互斥体启用了优先级继承选项，并且一个试图获得该互斥体的线程因此被提高了其优先级。该线程将在什么时候将其优先级恢复到其原始值？

7. 前一个问题中的线程是否有可能在它处于挂起状态时改变其优先级？如果是这样，那么可能会出现哪些问题呢？是否有任何情况可以证明执行这一行动？

8. 假设你的任务是创建一个看门狗线程，试图检测和纠正致命的拥抱。一般来说，描述如何完成此任务。

9. 描述tx_mutex_prioritize服务的目的，并给出一个示例。

10. 讨论您可以帮助避免优先级反转问题的两种方法。

11. 讨论两种方法来帮助你避免致命的拥抱问题。

12. 考虑图78，它包含了示例系统的部分活动跟踪。什么时候快速线程什么时候会领先慢线程？

第七章

内存管理：字节池和块池

介绍

回想一下，我们在上一章中为线程堆栈使用了数组。虽然这种方法具有简单性的优点，但它通常是不受欢迎的，而且相当不灵活。本章重点介绍了两个Azure RTOS ThreadX内存管理资源，它们提供了大量的灵活性：内存字节池和内存块池。

内存字节池是一个连续的字节块。在这样的池中，可以使用和重用任何大小（取决于池的总大小）的字节组。内存字节池是灵活的，可以用于线程堆栈和其他需要内存的资源。但是，这种灵活性会导致一些问题，例如在使用不同大小的字节组时，内存字节池的碎片化。

内存块池也是一个连续的字节块，但它被组织成一个固定大小的内存块的集合。因此，从内存块池中使用或重用的内存量总是相同的——一个固定大小的内存块的大小。没有碎片化问题，并且分配和释放内存块的速度很快。通常，使用内存块池优于使用内存字节池。

我们将在本章中研究和比较这两种类型的内存管理资源。我们将考虑每种类型的特性、功能、陷阱和服务。我们还将使用这些资源创建说明性的示例系统。

内存字节池的摘要

内存字节池类似于标准的C堆。[32]与C堆相比，Azure RTOS ThreadX应用程序可能使用多个内存字节池。此外，线程可以挂起到内存字节池上，直到所请求的内存可用为止。

来自内存字节池的分配类似于传统的malloc调用，其中包括所需的内存量（以字节为单位）。Azure RTOS ThreadX以第一次匹配的方式从内存字节池中分配内存，也就是说，它使用足够大到可以满足请求的第一个可用内存块。Azure RTOS ThreadX将此块中的多余内存转换为一个新块，并将其放回空闲内存列表中。*这个过程被称为碎片化。*

当Azure RTOS ThreadX对足够大的可用内存块执行后续的分配搜索时，它会将相邻的可用内存块合并在一起。*这个过程被称为碎片整理。*

每个内存字节池都是一个公共资源；Azure RTOS ThreadX对内存字节池的使用方式没有任何限制。[33]应用程序可以在初始化期间或在运行时期间创建内存字节池。对于应用程序可能使用的内存字节池的数量没有明确的限制。

内存字节池中可分配的字节数略小于创建过程中指定的数量。这是因为对空闲内存区域的管理引入了一些开销。池中的每个可用内存块都需要相当于两个开销的C指针。此外，当创建池时，Azure RTOS ThreadX会自动将其分成两个块，一个大的空闲块和一个位于内存区域末端的小的永久分配块。该被分配的端块用于提高分配算法的性能。它消除了在合并过程中不断检查池区域的末端的需要。在运行时期间，池中的开销量通常会增加。这部分是因为当分配了奇数字节时，Azure RTOS ThreadX会填充块以确保正确对齐

[32] 在C语言中，堆是一个内存区域，程序可以使用它来存储可变数量的数据，直到程序运行时才知道这些数据。

[33] 但是，不能从中断服务例程中调用内存字节池服务。（本主题将在下一章中进行讨论。）

的下一个内存块。此外，随着池变得更加分散，开销也会增加。

在创建期间指定内存字节池的内存区域。与其他内存区域一样，它也可以位于目标地址空间中的任何位置。这是一个重要的特性，因为它给应用程序提供了相当大的灵活性。例如，如果目标硬件具有高速存储区域和低速存储区域，那么用户可以通过在每个区域中创建一个池来管理这两个区域的内存分配。

应用程序线程可以在等待池中的内存字节时挂起。当有足够的连续内存可用时，挂起的线程接收其请求的内存并恢复。如果多个线程已挂起在同一内存字节池上，则Azure RTOS ThreadX将给它们提供内存，并按照它们在挂起线程列表（通常是FIFO）上出现的顺序恢复它们。但是，应用程序可以通过在解除线程暂停的字节释放调用之前调用tx_byte_pool_prioritize，从而导致挂起线程的优先级恢复。字节池优先级服务将最高优先级的线程放在挂起列表的前面，同时保留所有其他挂起的线程在相同的FIFO顺序中。

“现场的描述》

tx_byte_pool_id	字节池ID
tx_byte_pool_name	指向字节池名称的指针
tx_byte_pool_available	可用字节数
tx_byte_pool_fragments	池中的片段数
tx_byte_pool_list	字节池的头指针
tx_byte_pool_search	用于搜索内存的指针
tx_byte_pool_start	字节池区域的起始地址
tx_byte_pool_size	字节池大小（以字节为单位）
* tx_byte_pool_owner	在搜索过程中指向字节池所有者的指针
* tx_byte_pool_suspension_list	字节池暂停列表头
tx_byte_pool_suspended_count	挂起的线程数
* tx_byte_pool_created_next	指向创建列表中下一个字节池的指针
* tx_byte_pool_created_previous	指向已创建列表中的上一个字节池的指针

图81。内存字节池控制块

内存字节池控制块

每个内存字节池的特征可以在其控制块中找到。[34]它包含一些有用的信息，如池中的可用字节数。内存字节池控制块可以位于内存中的任何地方，但最常见的方法是通过在任何函数的范围之外定义它，使控制块成为全局结构。图81包含了包含此控制块的许多字段。

在大多数情况下，开发人员可以忽略内存字节池控制块的内容。但是，在调试过程中可能有几个字段很有用，例如可用的字节数、片段数以及挂起在此内存字节池上的线程数。

内存字节池的缺陷

尽管内存字节池提供了最灵活的内存分配，但它们也存在某种不确定性行为的问题。例如，一个内存字节池可能有2,000字节的可用内存，但不能满足甚至1,000字节的分配请求。这是因为不能保证有多少空闲字节是连续的。即使存在一个1,000字节的空闲块，也不能保证它可能需要多长时间才能找到该块。分配服务很可能必须搜索整个内存池，以找到1,000字节的块。由于这个问题，通常最好避免在需要确定性、实时行为的区域中使用内存字节服务。许多这样的应用程序在初始化或运行时配置时预先分配了它们所需的内存。另一种选择是使用内存块池（本章后面将进行讨论）。

分配的内存的用户不能在其边界之外写入。如果发生这种情况，则相邻（通常是后续的）内存区域发生损坏。其结果是不可预测的，而且往往是灾难性的。

内存字节池服务的摘要

附录B包含有关内存字节池服务的详细信息。本附录包含关于每个服务的信息，例如原型，一个摘要

[34] 内存字节池控制块的结构是在tx_api中定义的。h文件。

对服务的描述、所需的参数、返回值、注释和警告、允许的调用，以及一个显示如何使用该服务的示例。图82包含了所有可用的内存字节池服务的列表。在本章的后续章节中，我们将调查每个这些服务。

我们将首先考虑tx_byte_pool_create服务，因为它必须在任何其他服务之前被调用。

内存字节池服务	描述
tx_byte_allocate	分配内存字节
tx_byte_pool_create	创建一个内存字节池
tx_byte_pool_delete	删除一个内存字节池
tx_byte_pool_info_get	检索有关内存字节池的信息
tx_byte_pool_prioritize	确定内存字节池挂起列表的优先级
tx_byte_release	将字节释放回内存字节池

图82。内存字节池的服务

创建内存字节池

内存字节池用TX_BYTE_POOL数据类型声明，并使用tx_byte_pool_create服务定义。在定义内存字节池时，您需要指定其控制块、内存字节池的名称、内存字节池的地址以及可用的字节数。图83包含了这些属性的列表。

内存字节池控制块
内存字节池名称
内存字节池的位置
可用于内存字节池的字节总数

图83。内存字节池的属性

我们将开发一个创建内存字节池的示例来说明此服务的使用。我们将给我们的内存字节池命名为“my_pool”。图84包含了一个创建内存字节池的示例。

```
UINT状态;
TX_BYTE_POOL我_pool;

/*创建一个内存池，其地址从地址0x500000开始，总大小为2000字节。*/状态

=tx_byte_pool_create（&my_pool，“my_pool”，（VOID *）0x500000,2000）;

/*如果状态等于TX_SUCCESS，则my_pool可用于分配内存。*/
```

图84。正在创建内存字节池

如果变量状态包含返回值TX_SUCCESS，则已成功创建一个名为my_poop的内存字节池，它包含2000字节，从位置03500000开始。

从内存字节池中进行分配

在声明和定义了一个内存字节池之后，我们就可以开始在各种应用程序中使用它了。tx_byte_allocate服务是从内存字节池中分配内存字节的方法。要使用此服务，我们必须指示需要多少字节，以及如果此字节池中没有足够的内存，则如何操作。图85显示了一个示例分配，如果没有足够的内存，它将“永远等待”。如果分配成功，则指针memory_ptr将包含已分配字节的起始位置。

```
TX_BYTE_POOL我_pool;
未加符号的字符*memory_ptr; UINT状态;

/*从my_pool中分配一个112字节的内存区域。假设字节池具有
  已经通过调用tx_byte_pool_create创建了。*/

状态= tx_byte_allocate（&my_pool，VOID **）和memory_ptr，
                        112, tx_wait_forever );

/*如果状态等于TX_SUCCESS，则memory_ptr包含的地址
   分配的内存区域。*/
```

图85。从内存字节池中分配字节

如果变量状态包含返回值TX_SUCCESS，则已成功创建了由memory_ptr指向的112字节的块。

请注意，此服务所需的时间取决于块的大小和内存字节池中的碎片量。因此，您不应该在执行时间关键的线程期间使用此服务。

删除内存字节池

可以使用tx_byte_pool_delete服务删除内存字节池。所有因为等待此字节池的内存而挂起的线程都将被恢复，并接收TX_DELETED返回状态。图86显示了如何删除内存字节池。

```
TX_BYTE_POOL我_pool;
UINT状态;
…
/*删除整个内存池。假设池中已经有了
  是通过调用tx_byte_pool_create而创建的。*/

状态= tx_byte_pool_delete（&my_pool）；

/*如果状态为TX_SUCCESS，则会删除内存池。*/
```

图86。删除内存字节池

如果变量状态包含返回值TX_SUCCESS，则内存字节池已被成功删除。

正在检索内存字节池信息

有三种服务使您能够检索有关内存字节池的重要信息。第一个内存字节池的服务——tx_byte_pool_info_get服务—从内存字节池控制块检索信息子集。此信息提供了在特定时刻，即调用服务时的“快照”。另外两个服务提供了基于运行时性能数据的收集的汇总信息。其中一个服务——tx_byte_pool_performance_info_get服务——在调用服务之前为特定内存字节池提供信息摘要。相比之下

tx_byte_pool_performance_system_info_get检索在调用服务之前系统中所有内存字节池的信息摘要。这些服务对于分析系统的行为和确定是否存在潜在的问题领域很有用。tx_byte_pool_info_get[4]服务检索有关内存字节池的各种信息。检索到的信息包括字节池名称、可用的字节数、内存片段的数量、第一个在此字节池的挂起列表中的线程的位置、当前在此字节池上挂起的线程数以及下一个创建的内存字节池的位置。图87显示了如何使用此服务来获取有关内存字节池的信息。

```
TX_BYTE_POOL我_pool;
CHAR名称;
ULONG可用;
ULONG片段;
TX_THREAD*first_暂停;
ULONG暂停_count;
TX_BYTE_POOL *next_pool;
UINT状态;
…
/*检索有关以前创建的块池“my_pool”的信息。*/

状态=tx_byte_pool_info_get（我的池，名称，名称，可用，片段，首先暂停，暂停计
                              数，下一个池）；

/*如果状态为TX_SUCCESS，则所请求的信息有效。*/
```

图87。正在检索有关内存字节池的信息

如果变量状态包含返回值TX_SUCCESS，则已成功获得有关内存字节池的有效信息。

确定内存字节池暂停列表的优先级

当线程因为正在等待内存字节池而挂起时，它将以FIFO方式放置在挂起列表中。当内存字节池恢复足够的内存时，挂起列表中的第一个线程（不论优先级）将获得从该内存字节池中分配字节的机会。tx_byte_pool_prioritize服务将最高优先级的线程挂起为

位于挂起列表前面的特定内存字节池的所有权。所有其他线程保持挂起时的FIFO顺序。图88显示了如何使用此服务。

如果变量状态包含值TX_SUCCESS，则操作成功：挂起列表中最高优先级的线程已放在挂起列表的前面。如果此内存字节池上没有暂停线程，则服务还会返回TX_SUCCESS。在这种情况下，挂起列表保持不变。

```
TX_BYTE_POOL我_pool;
UINT状态;
…
/*确保最高优先级的线程将从这个池接收下一个空闲内存。*/

状态= tx_byte_pool_prioritize（&my_pool）;

/*如果状态为TX_SUCCESS，则最高优先级的挂起线程在的前面
  列表。如果有足够的内存来满足其请求*/，下一个tx_byte_release调用将唤醒这个
  线程
```

图88。确定内存字节池挂起列表的优先级

将内存释放到字节池

tx_byte_release服务将先前分配的内存区域释放回其关联的池。如果在这个池上挂起一个或多个线程，每个挂起的线程都会接收其请求的内存并恢复，直到池的内存耗尽或没有再挂起的线程。将内存分配给挂起线程的过程始终从挂起列表上的第一个线程开始。图89显示了如何使用此服务。

```
未加符号的字符*memory_ptr; UINT状态;
…
/*将内存释放回我的_pool。假设内存区域之前已从my_pool中分配。
  */

状态=tx_byte_release（（无效*）memory_ptr）;
/*如果状态等于TX_SUCCESS，则所指向的内存
  memory_ptr已返回到池中。*/
```

图89。将字节释放回内存字节池

如果变量状态包含值TX_SUCCESS，则memory_ptr指向的内存块已返回到内存字节池。

内存字节池示例-分配线程堆栈

在上一章中，我们使用数组来为线程堆栈提供内存空间。在本例中，我们将使用一个内存字节池来为这两个线程提供内存空间。第一步是声明线程和内存字节池：

```
TX_THREAD快速线程，慢线程；
TX_MUTEX我_mutex；
#定义STACK_SIZE 1024；
TX_BYTE_POOL我_pool；
```

在定义线程之前，我们需要创建内存字节池，并为线程堆栈分配内存。下面是字节池的定义，由4,500个字节组成，从位置03500000开始。

```
UINT状态；
社会地位=tx_byte_pool_create（&我的_pool，“我的_pool”，（无效*）
                                        03500000,4500）；
```

假设返回值为TX_SUCCESS，我们已经成功地创建了一个内存字节池。接下来，我们从这个字节池为spepy_slade堆栈分配内存，如下所示：

```
字*stack_ptr；
社会地位=tx_byte_allocate(&我的_pool
                                （无效**）&stack_ptr，STACK_SIZE，
                                TX_WAIT_FOREVER)；
```

假设返回值为TX_SUCCESS，我们已经成功地为堆栈分配了一个内存块，这是由stack_ptr指向的。接下来，我们使用此内存块为其堆栈定义speedy_shaed（代替上一章中使用的数组stack_speedy），如下所示：

```
tx_thread_create(和“快速线”，
                    Speedy_Thread_条目，0
                    stack_ptr , STACK_SIZE ,
                    5, 5, tx_no_time_slice , tx_auto_start );
```

我们以类似的方式定义慢线程。线程输入功能保持不变。

内存字节池内部

当TX_BYTE_POOL数据类型用于声明一个字节池时，将创建一个字节池控制块，并将该控制块添加到一个双链接的循环列表中，如图90所示。

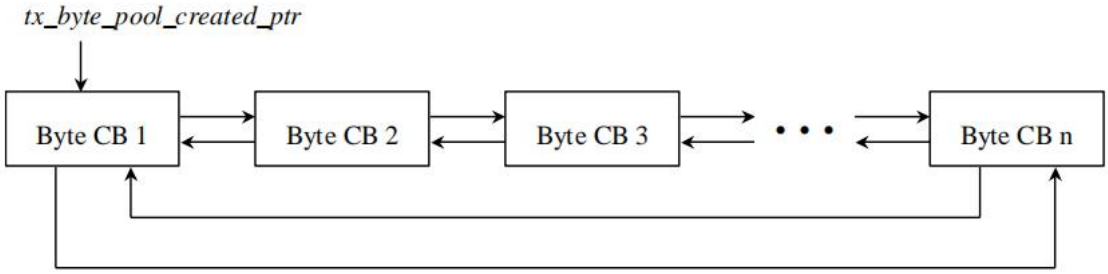


图90。已创建的内存字节池列表

名为tx_byte_pool_created_ptr的指针指向列表中的第一个控制块。有关字节池属性、值和其他指针，请参见字节池控制块中的字段。

来自内存字节池的分配类似于传统的malloc调用，其中包括所需的内存量（以字节为单位）。Azure RTOS ThreadX以首次匹配的方式从池中分配，将该块中的多余内存转换为一个新块，并将其放回空闲内存列表中。*这个过程被称为碎片化。*

Azure RTOS ThreadX在后续的分配搜索足够大的空闲内存块时将空闲内存块合并在一起。*这个过程被称为碎片整理。*

内存字节池中可分配的字节数略小于创建过程中指定的数量。这是因为对空闲内存区域的管理引入了一些开销。池中的每个可用内存块都需要相当于两个开销的C指针。此外，当创建池时，Azure RTOS ThreadX会自动分配两个块，一个大的空闲块和一个小的永久分配的块。这个已分配的端块用于改进

分配算法的性能。它消除了在合并过程中不断检查池区域的末端的需要。

在运行时期间，池中的开销量通常会增加。这部分是因为当分配了奇数字节时，Azure RTOS ThreadX会屏蔽所分配的块，以确保下一个内存块的正确对齐。此外，随着池变得更加分散，开销也会增加。

图91包含内存字节池创建之后，但在任何内存分配之前的说明。

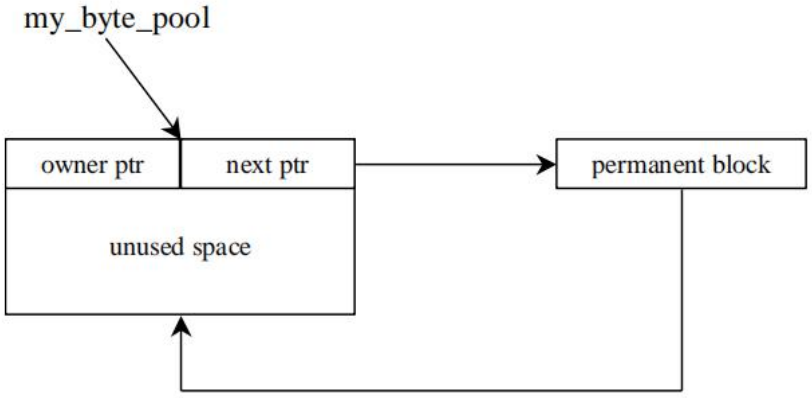


图91。创建时的内存字节池的组织结构

最初，所有可用的内存空间都被组织成一个连续的字节块。但是，来自这个字节池的每个连续分配都可能细分可用的内存空间。例如，图92显示了第一次内存分配后的一个内存字节池。

内存块池的汇总表

在实时应用程序中，以快速和确定性的方式分配内存是必不可少的。这是通过创建和管理多个被称为内存块池的固定大小的内存块池来实现的。

因为内存块池由固定大小的块组成，因此使用它们不涉及碎片化问题。这是至关重要的，因为碎片化会导致一些行为

固有的不确定性。此外，分配和释放固定大小的块是快速的——所需的时间与简单的链表操作相当。此外，当分配服务从内存块池中进行分配和释放时，它不必搜索块列表——它总是在可用列表的顶部进行分配和释放。这提供了尽可能快的链表处理，并可能有助于将当前使用的内存块保持在缓存中。

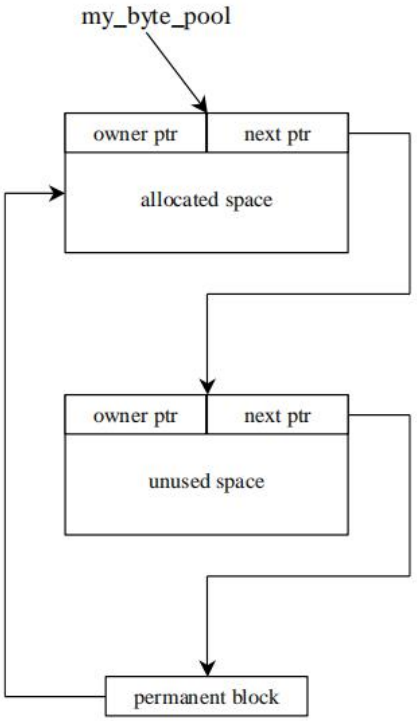


图92。第一次分配后的内存字节池

缺乏灵活性是固定大小的内存池的主要缺点。池的块大小必须足够大，以处理其用户最坏情况下的内存需求。从同一个池发出许多不同大小的内存请求可能会导致内存浪费。一种可能的解决方案是创建多个包含不同大小的内存块的不同内存块池。

每个内存块池都是一个公共资源。Azure RTOS ThreadX没有对如何使用池施加任何约束。应用程序可以在初始化期间或在运行期间从应用程序线程内创建内存块池。对应用程序可能使用的内存块池的数量没有限制。

如前所述，内存块池包含许多固定大小的块。在创建池期间指定块大小的字节。池中的每个内存块都会施加少量的开销——C指针的大小。此外，Azure RTOS ThreadX可以填充块的大小，以保持每个内存块的开始正确对齐。

池中的内存块的数量取决于块的大小和创建期间提供的内存区域中的总字节数。要计算池的容量（将可用的块数），请将块大小（包括填充和指针开销字节）划分为所提供的内存区域中的总字节数。

块池的内存区域是在创建期间指定的，并且可以位于目标地址空间中的任何位置。这是一个重要的特性，因为它给应用程序提供了相当大的灵活性。例如，假设一个通信产品具有I/O的高速内存区域。可以通过设置内存块池轻松管理此内存区域。

应用程序线程可以在等待来自空池中的内存块时挂起。当一个块返回到池时，Azure RTOS ThreadX将此块提供给挂起的线程并恢复该线程。如果多个线程挂起在同一内存块池上，Azure RTOS ThreadX将按照它们出现在挂起线程列表上的顺序（通常是FIFO）的顺序恢复它们。

但是，应用程序也可以导致恢复最高优先级的线程。为了实现这一点，应用程序在解除线程暂停的块释放调用之前调用tx_block_pool_prioritize。块池具有优先级，服务将最高优先级的线程放在挂起列表的前面，同时将所有其他挂起的线程保留在相同的FIFO顺序中。

内存块存储池控制块

每个内存块池的特征可以在其控制块中找到。[35]它包含诸如块大小和剩余的内存块数量等信息。内存池控制块可以位于内存中的任何位置，但它们通常被定义为任何函数范围之外的全局结构。图93列出了内存池控件块的大多数成员。

场地	描述
tx_block_pool_id	块池ID
tx_block_pool_name	指向阻止池名称的指针
tx_block_pool_available	可用块数
tx_block_pool_total	池中的初始块数
tx_block_pool_available_list	可用块池的头指针
tx_block_pool_start	块池内存区域的起始地址
tx_block_pool_size	块池的字节大小
tx_block_pool_block_size	每个内存块的大小-四舍五入
* tx_block_pool_suspension_list	阻止池暂停列表头
tx_block_pool_suspended_count	挂起的线程数
* tx_block_pool_created_next	指向创建列表中下一个块池的指针
* tx_block_pool_created_previous	指向已创建列表中的上一个块池的指针

图93。内存块存储池控制块

已分配的内存块的用户不能在其边界之外写入。如果发生这种情况，则在相邻的（通常是后续的）内存区域中就会发生损坏。其结果是不可预测的，而且往往是灾难性的。

在大多数情况下，开发人员可以忽略内存块池控制块的内容。但是，在调试过程中，有几个字段可能很有用，例如可用块的数量、初始块数、实际块的大小、块池中的总字节数以及在此内存块池上挂起的线程数。

[35] 内存块池控制块的结构已在tx_api中定义。h文件。

内存块池服务的摘要

附录A包含有关内存块池服务的详细信息。本附录包含关于每个服务的信息，例如原型、服务的简要描述、所需参数、返回值、注释和警告、允许的调用，以及显示如何使用该服务的示例。图94包含了所有可用的内存块池服务的列表。在本章的后续章节中，我们将调查每个这些服务。

内存块池
服务的描述

tx_block_allocate	分配一个固定大小的内存块
tx_block_pool_create	创建一个固定大小的内存块池
tx_block_pool_delete	删除内存块池
tx_block_pool_info_get	检索有关内存块池的信息
tx_block_pool_prioritize	确定内存块池挂起列表的优先级
tx_block_release	将一个固定大小的内存块释放回池中

图94。内存块池的服务

我们将首先考虑tx_block_pool_create服务，因为它必须在任何其他服务之前被调用。

正在创建内存块池

内存块池用TX_BLOCK_POOL数据类型声明，并使用tx_block_pool_create服务定义。在定义内存块池时，您需要指定它的控制块、内存块池的名称、内存块池的地址以及可用的字节数。图95包含了其中的一个列表
属性

内存块存储池控制块
内存块池的名称
每个固定大小的内存块中的字节数

内存块池的起始地址
块池可用的字节总数

图95。内存块池属性

我们将开发一个创建内存块池的示例来说明此服务的使用，我们将将其命名为“my_pool”。图96包含了一个创建内存块池的示例。

```
TX_BLOCK_POOL我_pool;
UINT状态;

/*创建一个内存池，从地址0x100000开始，其总大小为1000字节。这个池中的
  每个块都被定义为50个字节长。*/

状态=tx_block_pool_create(&我的_pool，“我的_pool_nome”，
                          50、（VOID *）0x100000、1000);

/*如果状态等于TX_SUCCESS，则my_pool包含18个内存块，每个内存块包含50个
  字节。池中没有20个块的原因是与每个块*/相关联的一个开销指针
```

图96。正在创建内存块池

如果变量状态包含返回值TX_SUCCESS，那么我们已经成功地创建了一个名为my_pool的内存块池，它总共包含1,000个字节，每个块包含50个字节。块数计算如下：

$$\text{块的总数} = \frac{\text{合计数量的字节可用}}{\text{（每个内存块中的字节数）+（大小（void *））}}$$

假设（void*）的值为4字节，则计算可用的块总数：

$$\text{Total Number of Blocks} = \frac{1000}{(50)+(4)} = 18.52 = 18 \text{ blocks}$$

使用上述公式可避免在内存块池中浪费空间。请务必仔细估计所需的块大小和池的可用内存量。

分配内存块池

在声明和定义了内存块池之后，我们就可以开始在各种应用程序中使用它了。tx_block_allocate服务是从内存块池中分配一个固定大小的内存块的方法。因为内存块池的大小是在创建时确定的，所以如果这个块池没有足够的内存，我们需要指示该怎么做。图97包含了一个从内存块池中分配一个块的示例，如果没有足够的内存可用，我们将“永远等待”。在内存分配成功后，指针memory_ptr包含已分配的固定大小的内存块的起始位置。

```
TX_BLOCK_POOL我_pool;
无符号字符*memory_ptr;
UINT状态;
…
/*从my_pool中分配一个内存块。假定
  池已经通过调用来创建了
  tx_block_pool_create .*/

状态= tx_block_allocate（&my_pool，（VOID **）&memory_ptr
                          ，TX_WAIT_FOREVER）;

/*如果状态等于TX_SUCCESS，则memory_ptr包含
  已分配的内存块的地址。*/
```

图97。分配一个固定大小的内存块

如果变量状态包含返回值TX_SUCCESS，那么我们已经成功地分配了一个固定大小的内存块。这个方块是由memory_ptr所指向的。

正在删除内存块池

可以使用tx_block_pool_delete服务删除内存块池。所有因为等待此块池的内存而挂起的线程都将恢复，并接收TX_DELETED返回状态。图98显示了如何删除内存块池。

如果变量状态包含返回值TX_SUCCESS，那么我们已成功地删除了内存块池。

```
TX_BLOCK_POOL我_pool;
UINT状态;
…
/*删除整个内存块池。假设已经通过调用tx_block_pool_create而创建了该池
。*/

状态= tx_block_pool_delete（&my_pool）;
/* If状态为== TX_SUCCESS，则内存块池已被删除。*/
```

图98。正在删除内存块池

正在检索内存块池信息

tx_block_pool_info_get服务检索有关内存块池的各种信息。检索到的信息包括块池名称、可用块数、池中的块总数、此块池的挂起列表中的线程的位置、当前此块池上挂起的线程数以及下一个创建的内存块池的位置。图99显示了如何使用此服务来获取有关内存块池的信息。

```
TX_BLOCK_POOL我_pool;
CHAR名称;
ULONG可用;
ULONG total_block;
TX_THREAD*first_暂停;
ULONG暂停_count;
TX_BLOCK_POOL *next_pool;
UINT状态;
…
/*检索有关以前创建的块池“my_pool”的信息。*/

状态=tx_block_pool_info_get（&我的池，名称，可用，总块，第一次暂停，第一次暂停
                                ，第一次计数，下一个池）;
/*如果状态为TX_SUCCESS，则该信息请求有效。*/
```

图99。正在检索有关内存块池的信息

如果变量状态包含返回值TX_SUCCESS，则我们已成功获得了有关内存块池的有效信息。

确定内存块池暂停列表的优先级

当线程因为正在等待内存块池而挂起时，它将以FIFO方式放置在挂起列表中。当内存块池恢复一个内存块时，挂起列表中的第一个线程（不论优先级）将获得从该内存块池中获取块的机会。tx_block_pool_prioritize服务将最高优先级的线程挂起的线程放置在挂起列表的前面。所有其他线程保持挂起时的FIFO顺序。图100包含了一个显示如何使用此服务的示例。

```
TX_BLOCK_POOL我_pool;
UINT状态;
…
/*确保将接收到最高优先级的线程
  这个池子中的下一个免费方块。*/

状态= tx_block_pool_prioritize（&my_pool）；

/*如果状态为TX_SUCCESS，则最高优先级的挂起线程位
  于列表的前面。这个
  下一个tx_block_release调用将唤醒这个线程。*/
```

图100。确定内存块池挂起列表的优先级

如果变量状态包含值TX_SUCCESS，则优先级排序请求已成功。挂起列表中等待名为“my_pool”的内存块池的最高优先级线程已移动到挂起列表的前面。如果没有线程在等待这个内存块池，那么服务调用也会返回TX_SUCCESS。在这种情况下，暂停列表保持不变。

```
TX_BLOCK_POOL我_pool;
无符号字符*memory_ptr;
UINT状态;
…
/*将一个内存块释放回my_pool。假设已创建了池，并且已分配了
  内存块。*/

状态=tx_block_release（（无效*）memory_ptr）；
```

```
/*如果状态为TX_SUCCESS，则memory_ptr指向的内存块已返回到池
  */
```

图101。将一个块释放到内存块池中

释放内存块

tx_block_release服务将一个先前分配的内存块释放回其关联的块池。如果在此池上挂起一个或多个线程，每个挂起线程将接收一个内存块并恢复，直到池耗尽块或不再有挂起线程。将内存分配给挂起线程的过程始终从挂起列表上的第一个线程开始。图101显示了如何使用此服务。

如果变量状态包含值TX_SUCCESS，则memory_ptr指向的内存块已返回到内存块池。

内存块池示例-分配线程堆栈

在上一章中，我们从数组分配线程堆栈内存，在本章的前面中，我们从字节池中分配线程堆栈。在此示例中，我们将使用一个内存块池。第一步是声明线程和内存块池：

```
TX_THREAD快速线程，慢线程；
TX_MUTEX我_mutex；
#定义STACK_SIZE 1024；
TX_BLOCK_POOL我_pool；
```

在定义线程之前，我们需要创建内存块池，并为线程堆栈分配内存。下面是块池的定义，它由四个每个1,024字节的块组成，从位置03500000开始。

```
UINT状态；
社会地位=tx_block_pool_create（&my_pool，“my_pool”，1024，（无效*
                                        ）03500000,4520）；
```

假设返回值为TX_SUCCESS，那么我们已经成功地创建了一个内存块池。接下来，我们从该块池为spepe_thade堆栈分配内存，如下所示：

```
字*stack_ptr；
```

```
社会地位=tx_block_allocate(&我的_pool
                    （VOID **）&stack_ptr，
                   tx_wait_forever );
```

假设返回值为TX_SUCCESS，我们已经成功地为堆栈分配了一个内存块，这是由stack_ptr指向的。接下来，我们通过使用其堆栈的内存块来定义spedy_slaed，如下所示：

```
tx_thread_create(和“快速线”，
                    Speedy_Thread_条目，0
                    stack_ptr , STACK_SIZE ,
                    5、5、TX_NO_TIME_SLICE、TX_AUTO_START)；我们以
```

类似的方式定义慢线程。线程输入功能保持不变。

内存块池内部构件

当TX_BLOCK_POOL数据类型用于声明块池时，将创建一个块池控制块，并且将该控制块添加到一个双链接的循环列表中，如图102所示。

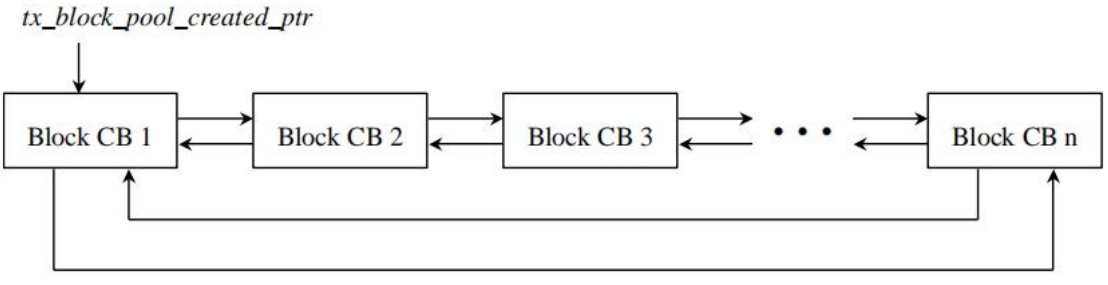


图102。已创建的内存阻止池列表

名为tx_block_pool_created_ptr的指针指向列表中的第一个控制块。有关块池属性、值和其他指针，请参见块池控制块中的字段。

如前所述，块池包含固定大小的内存块。这种方法的优点包括快速分配和发布块，并且没有碎片化问题。一个可能的缺点是，如果块的大小太大，可能会浪费空间。然而，开发人员可以通过创建多个块池来最小化这个潜在的问题

有不同的块大小。池中的每个块都需要少量的开销，即一个所有者指针和下一个块指针。图102说明了内存块池的组织结构。

概述和比较

在本章中，我们考虑了两种进行内存管理的方法。第一种方法是内存字节池，它允许使用和重用可变大小的字节组。这种方法具有简单性和灵活性的优点，但会导致碎片化问题。由于碎片化，内存字节池可能有足够的总字节来满足内存请求，但可能无法满足该请求，因为可用的字节并不连续。因此，我们通常建议您避免在确定性的实时应用程序中使用内存字节池。

内存管理的第二种方法是内存块池，它由固定大小的内存块组成，从而消除了碎片化问题。内存块池失去了一些灵活性，因为所有内存块的大小都相同，而且给定的应用程序可能不需要那么大的空间。但是，开发人员可以通过创建多个内存块池来缓解这个问题，每个内存块池具有不同的块大小。此外，分配和释放内存块是快速的和可预测的。一般来说，我们建议在确定性的实时应用程序中使用内存块池。

主要术语和短语

存储器分配
块尺寸计算
创建内存池
消除碎化
删除内存池
内存字节池
内存字节池控制块指针开销
优先级存储池挂起列表
固定尺寸的块
分裂
信息检索
存储器块池
内存块池控制块释放内存
暂停时等待内存线程堆栈内存
分配等待选项

问题

1. 内存块池被推荐用于确定性的实时应用程序。在什么情况下，您应该使用内存字节池？

2. 在上一章中，线程堆栈是从数组中分配出来的。当为线程堆栈提供内存时，内存块池比数组有什么优势？

3. 假设一个应用程序已经创建了一个内存字节池，并已经创建了几个分配。然后，当池总共有500个字节可用时，应用程序会请求200个字节。解释为什么池可能不能及时满足这个要求。

4. 假设创建的总计为1000字节的内存块池，每个块的大小为100字节。解释为什么这个池包含的块少于10个块。

5. 在上一个问题中创建内存块池，并检查控制块中的以下字段：tx_block_pool_available、tx_block_pool_size、tx_block_pool_block_size和tx_block_pool_suspended_count。

6. *该部分标题为WamoJ(：/o0入doo/3xemd/a-Y//o0e！！e0入s包含一个使用其堆栈的内存块的快速线程的定义。*使用其堆栈的内存块为slow_线程开发一个定义，然后构建和调试生成的系统。

第8章

内部系统时钟和

应用程序计时器

介绍

快速响应异步的外部事件是实时、嵌入式应用程序最重要的功能。然而，许多这些应用程序还必须在预定的时间间隔内执行某些活动。

Azure RTOS ThreadX应用程序计时器允许您在特定的时间间隔内执行应用程序C函数。您还可以将应用程序计时器设置为只过期一次。*这种类型的计时器被称为oUa-sVo!*!!maJ，而重复的间隔计时器被称为daJ！操作！？*!!maJs.*每个应用程序计时器都是一个公共资源。

时间间隔是通过周期性的定时器中断来测量的。每个计时器中断被称为计时器计时器。计时器标记之间的实际时间是由应用程序指定的，但是10 ms是许多实现的规范。[36]

底层硬件必须能够产生周期性的中断，以便应用程序定时器能够正常工作。在某些情况下，处理器具有内置的周期性中断能力。如果没有，用户的计算机板必须有一个能够产生周期性中断的外围设备。

Azure RTOS ThradX即使没有周期性中断源也可以工作。但是，所有与计时器相关的处理都将被禁用。这包括时间切片、暂停超时和计时器服务。

[36] 定期定时器设置通常可以在tx_i11汇编文件中找到。

计时器过期间隔是根据计时器标记来指定的。计时器计数从指定的过期值开始，每次计时器计时减少1。因为应用程序计时器可以在计时器中断（或计时器计时器）之前启用，所以计时器可以提前过期一次。

如果计时率为10 ms，应用程序计时器可能提前过期10 ms。这种不准确性对于10个ms计时器比对于1秒计时器更重要。当然，增加计时器中断频率会降低这一误差幅度。

应用程序计时器按其处于活动状态时的顺序执行。例如，如果您创建了三个具有相同过期值的计时器，然后激活它们，则它们对应的过期功能[37]保证按照您激活计时器的顺序执行。

默认情况下，应用程序计时器在优先级为0的隐藏系统线程中执行，这比任何应用程序线程都要高。因此，您应该保持处理内部计时器过期函数。重要的是要避免暂停来自应用程序计时器的过期功能中的任何服务调用。

同样重要的是，尽可能避免使用每个计时器过期的计时器。这可能会在应用程序中引起过多的开销。

除了应用程序计时器之外，Azure RTOS ThreadX还提供了一个单一的连续递增的32位滴答计数器。这个刻度计数器，或内部系统时钟，在每个计时器中断上增加一个。应用程序可以通过分别调用tx_time_get和tx_time_set来读取或设置这个32位计数器。内部系统时钟的使用完全由应用程序决定。

我们将首先考虑内部系统时钟的两个服务(i。e., tx_time_get和tx_time_set)，然后我们将研究应用程序计时器。

[37] 过期函数有时被称为超时函数。

内部系统时钟服务

Azure RTOS ThradX在应用程序初始化期间将内部系统时钟设置为零，每个计时器[38]使时钟增加1个。内部系统时钟仅供开发人员使用；Azure RTOS ThreadX不将其用于任何目的，包括实现应用程序计时器。应用程序可以在内部系统时钟上执行两个操作：读取当前时钟值，或设置其值。附录一包含了关于内部系统时钟服务的附加信息。

tx_time_get服务从内部系统时钟中检索当前时间。图103说明了如何使用此服务。

```
ULONG电流时间；

/*检索当前系统时间。*/

当前时间= tx_time_get（）；

/*变量当前时间现在包含当前系统时间*/
```

图103。从内部系统时钟获取当前时间

调用此服务后，变量purut_time包含内部系统时钟的副本。此服务可用于测量已运行的时间和执行其他与时间相关的计算。

tx_time_set服务将内部系统时钟的当前时间设置为某个指定的值。图104说明了如何使用此服务。

```
/*设置内部系统时间为0x1234。*/

tx_time_set (0x1234);

/*当前时间现在包含0x1234，直到下一个计时器中断*/
```

图104。设置内部系统时钟的当前时间

[38] ~~每个计时器标记所表示的实际时~~间是特定于应用程序的。

调用此服务后，内部系统时钟的当前时间包含值031234。时间将保持在这个值，直到下一个计时器标记，当它将增加1。

应用计时器控制块

每个应用计时器的特性可以在其应用计时器控制块中找到。[39]它包含一些有用的信息，如ID、应用程序计时器名称、剩余计时器标记的数量、重新初始化值、指向超时函数的指针、超时函数的参数以及各种指针。与其他控制块一样，Azure RTOS ThreadX禁止应用程序显式地修改应用程序计时器控制块。

应用程序计时器控制块可以位于内存中的任何地方，但最常见的方法是通过将控制块定义在任何函数的范围之外，从而将其变为全局结构。图105包含了包含此控制块的许多字段。

“现场的描述》

tx_timer_id	应用程序计时器ID
tx_timer_name	指向应用程序计时器名称的指针
tx_remaining_ticks	剩余计时器的数量
tx_re_initialize_ticks	重新初始化计时器-标记值
tx_timeout_function	指向超时功能的指针
tx_timeout_param	超时功能参数
tx_active_next	指向下一个活动的内部计时器的指针
tx_active_previous	指向先前活动的内部计时器的指针
tx_list_head	指向内部计时器列表负责人的指针
tx_timer_created_next	指向创建列表中下一个字节池的指针
tx_timer_created_previous	指向已创建列表中的上一个字节池的指针

图105。应用计时器控制块

当使用TX_TIMER数据类型声明应用程序计时器时，将创建一个应用程序计时器控制块。例如，我们声明my_timer如下：

TX_TIMER我_timer；

[39] 应用程序计时器控制块的结构是在tx_api中定义的。h文件。

应用程序计时器的声明通常出现在应用程序的声明和定义部分。

应用程序计时器服务的摘要

附录J包含了关于应用程序计时器服务的详细信息。本附录包含关于每个服务的信息，例如原型、服务的简要描述、所需参数、返回值、注释和警告、允许的调用，以及显示如何使用该服务的示例。图106包含了所有可用的应用程序计时器服务的列表。我们将在本章的后续章节中调查这些服务。

应用程序计时器服务	描述
tx_timer_activate	激活应用程序计时器
tx_timer_change	更改应用程序计时器的特性
tx_timer_create	创建应用程序计时器
tx_timer_deactivate	停用应用程序计时器
tx_timer_delete	删除应用程序计时器
tx_timer_info_get	检索有关应用程序计时器的信息

图106。应用程序计时器的服务

我们将首先考虑tx_timer_pool_create服务，因为它需要在任何其他服务之前被调用。

创建应用程序计时器

应用程序计时器用TX_TIMER数据类型声明，并使用tx_timer_create服务定义。定义应用程序计时器时，必须指定其控制块、应用程序计时器的名称、计时器到期时调用的过期函数、传递给过期函数的输入值、计时器到期前的初始计时器数、所有定时器过期的计时器数

在第一个之后，以及用于确定何时激活计时器的选项。初始计时数的有效范围为1到0x FFFFFFFF（包括）。

对于后续的时间计时器，值的有效范围从0到0x FFFFFFFF（包括），其中值0表示这是一次性计时器，该范围内的所有其他值都用于周期计时器。图107包含了这些属性的列表。

我们将用一个示例来说明应用程序计时器创建服务。我们将给我们的应用程序计时器的名称“my_timer”，并使它立即激活。计时器将在100个计时器后过期，调用名为“my计时器”的过期函数，然后每25个计时器继续这样做。图107包含了一个应用程序计时器创建的示例。

应用计时器控制块
应用程序计时器名称
在计时器到期时调用的到期功能
要传递到过期函数的输入
计时器的初始数量
第一个计时器后所有计时器到期的计时器标记数
自动激活选项

图107。应用程序计时器的属性

如果变量状态包含返回值TX_SUCCESS，则我们已成功创建了应用程序计时器。我们必须在我们的程序的声明和定义部分放置一个过期函数的原型，如下所示：

```
无效的我的_timer_函数（ULONG）；
```

过期函数定义出现在程序的最后一节中，其中定义了线程输入函数。下面是该函数定义的框架。

```
无效我的_timer_函数（ULONG无效）
{
:
}
```

激活应用程序计时器

当使用TX_NO_ACTIVATE选项创建应用程序计时器时，它将保持不活动状态，直到调用tx_timer_activate服务为止。类似地，如果应用程序计时器，它将保持不活动，直到调用tx_timer_activate服务。如果两个或多个应用程序计时器同时过期，则相应的过期功能将按其被激活时的顺序执行。图108包含了一个应用程序计时器激活的示例。

```
TX_TIMER我_timer;
UINT状态;
…
/*激活一个应用程序计时器。假设已创建了应用程序计时器。*/

状态= tx_timer_activate（&my_timer）;

/*如果状态等于TX_SUCCESS，则应用程序计时器现在为活动*/
```

图108。激活一个应用程序计时器

更改应用程序计时器

创建应用程序计时器时，必须在计时器到期前指定初始计时器数，以及第一次计时器到期后所有计时器过期的计时器数。调用tx_timer_change服务可以更改这些值。您必须在调用此服务之前停用应用程序计时器，并在此服务之后调用tx_timer_activate以重新启动计时器。图109说明了如何调用此服务。

```
TX_TIMER我_timer;
UINT状态;
…
/*更改一个以前创建的和现在停用的计时器
  每50个计时器点过期一次，包括初始过期时间。*/状态=
tx_timer_change（&my_timer，50,50）;
/*如果状态等于TX_SUCCESS，指定的计时器将更改为每50个计时器过期
  。*/
/*激活指定的计时器以重新启动它。*/
  状态= tx_timer_activate（&my_timer）;
```

图109。更改应用程序计时器的特性

如果变量状态包含返回值TX_SUCCESS，那么我们已经成功地将初始过期和后续过期的计时器标记数更改为50。

停用应用程序计时器

在修改应用程序计时器的定时特性之前，必须首先停用该定时器。这是tx_timer_deactivate服务的唯一目的。图110显示了如何使用此服务。

```
TX_TIMER我_timer;
UINT状态;
…
/*停用一个应用程序计时器。假设应用程序计时器具有
  已经创建。*/

状态= tx_timer_deactivate（&my_timer）；

/*如果状态等于TX_SUCCESS，则应用程序计时器现在已停用*/
```

图110。停用应用程序计时器

如果变量状态包含值TX_SUCCESS，则应用程序计时器现在将被停用。此计时器将处于非活动状态，直到被tx_timer_activate服务激活。

删除应用程序计时器

tx_timer_delete服务将删除一个应用程序计时器。图111显示了如何删除应用程序计时器。

```
TX_TIMER我_timer;
UINT状态;
…
/*删除应用程序计时器。假设应用程序计时器具有
  已经创建。*/

状态= tx_timer_delete（&my_timer）；

/*如果状态等于TX_SUCCESS，则删除应用程序计时器*/
```

图111。删除应用程序计时器

如果变量状态包含返回值TX_SUCCESS，则我们已成功删除了应用程序计时器。请确保您不会在无意中使用了已删除的计时器。

正在检索应用程序计时器信息

有三个服务允许您检索有关应用程序计时器的重要信息。针对应用程序计时器的第一个这样的服务——tx_timer_pool_info_get服务——从应用程序计时器控制块中检索信息的子集。此信息在特定时刻提供“快照”，即调用服务时提供的“快照”。另外两个服务提供了基于运行时性能数据的收集的汇总信息。其中一个服务——tx_timer_pool_performance_info_get服务——为调用服务时的特定应用程序计时器提供信息摘要。相比之下，tx_timer_pool_performance_system_info_get会检索到调用服务之前系统中所有应用程序计时器的信息摘要。这些服务对于分析系统的行为和确定是否存在潜在的问题领域很有用。tx_timer_info_get[40]服务检索有关应用程序计时器的各种信息。检索到的信息包括应用程序计时器名称、其活动/非活动状态、计时器过期前的计时器标记数、第一次过期后计时器过期的后续计时器数，以及指向下一个创建的应用程序计时器的指针。图112显示了如何使用此服务来获取有关应用程序计时器的信息。

如果变量状态包含返回值TX_SUCCESS，则我们检索了有关计时器的有效信息

。

[40]默认情况下，只启用了tx_timer_info_get服务。必须启用其他两个信息收集服务才能使用它们。