(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 111052094 A
(43)申请公布日 2020.04.21

(21)申请号 201880058139.8

(22)申请日 2018.09.06

(30)优先权数据
15/698,568 2017.09.07 US

(85)PCT国际申请进入国家阶段日
2020.03.06

(86)PCT国际申请的申请数据
PCT/US2018/049796 2018.09.06

(87)PCT国际申请的公布数据
WO2019/051120 EN 2019.03.14

(71)申请人 阿里巴巴集团控股有限公司
地址 英属开曼群岛大开曼资本大厦一座四层847号邮箱

(72)发明人 蒋晓维

(74)专利代理机构 北京清源汇知识产权代理事务所(特殊普通合伙) 11644
代理人 冯德魁 窦晓慧

(51)Int.Cl.
G06F 12/00(2006.01)

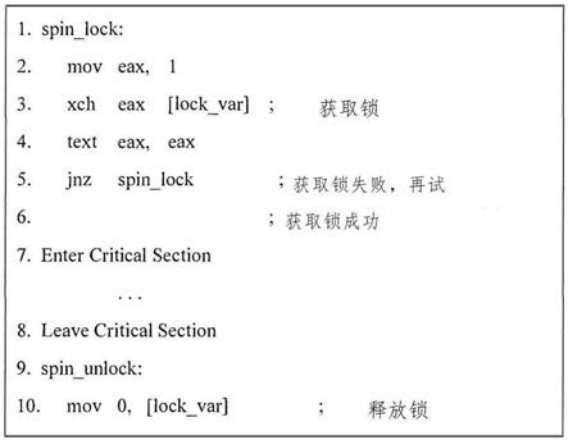
权利要求书2页 说明书7页 附图6页

(54)发明名称

使用C状态和睿频加速提高用户空间的自旋锁效率

(57)摘要

公开了用于通过使用内核结构的多个线程来有效保护同时访问用户空间共享数据的系统和方法。此外,还公开了一种在自旋锁内允许减少与改进的自旋锁相关的性能和功率干扰的机制。这使得可以通过提高运行线程的CPU核心的频率和电压来使临界区中的线程更快地完成执行。改进的自旋锁允许线程进入节能状态,并允许临界区指示PCU将专用的净空功率预算分配给执行该指令的核心。改进的自旋锁还可以提供CPU资源的时钟门控期间动态功率节省,以及在CPU资源的功率门控期间动态和静态功率节省。



CN 111052094 A

1.一种处理系统,包括:

存储器,所述存储器被配置为对应用程序的多个线程提供用户空间共享数据的访问;

多个核心,所述多个核心中的每一个被配置为执行多个线程中的一个或更多个线程,其中多个核心中的一个被配置为:

包括一获取锁的线程,该锁指示对共享数据的处理,和

生成核心已获取锁的通知,其中所述通知指示试图访问共享数据的一个或更多个其它线程进入节能状态。

2.如权利要求1所述的处理系统,其中所述获取锁进一步指示所述线程将进入或已进入临界区。

3.权利要求1和2中任一项的处理系统,进一步包括功率控制单元,所述功率控制单元被配置为基于进入临界区的线程向核心分配额外的功率预算。

4.如权利要求3所述的处理系统,其中所述功率控制单元被进一步配置为确定多个核心中的每个核心确定适当的P状态。

5.如权利要求3所述的处理系统,其中所述功率控制单元进一步被配置为检测多个具有已进入节能状态的线程的核心的功率降低。

6.如权利要求3所述的处理系统,其中所述功率控制单元进一步配置为增加具有已进入临界区的线程的内核的电压和频率。

7.如权利要求1-6中任一项所述的处理系统,其中,已进入节能状态的一个或更多个其它线程监视锁是否已释放。

8.权利要求7的处理系统,其中,所述一个或更多个其它线程基于对包含已获取锁的线程的核心存储器位置的一个或更多个观察来监控锁是否已被释放。

9.根据权利要求7所述的处理系统,其中,如果一个或更多个其它线程确定锁已被释放,一个或更多个其它线程中的至少一个线程尝试获取共享数据的锁。

10.根据权利要求1-9任一所述的处理系统,其中所述节能状态为选定的C状态。

11.一种计算机实现的方法,所述方法在具有处理器和多个核心的处理系统上执行,包括:

向应用程序的多个线程提供对用户空间的共享数据的访问;

通过多个核心执行多个线程的一个或更多个线程;

通过多个核心的其中一个核心,由线程获取指示用户空间中共享数据的处理的锁,以及

由多个核心中的核心生成核心已获得锁的通知,其中,所述通知指示一个或更多个线程尝试访问用户空间中的共享数据以进入节能状态。

12.根据权利要求11所述的方法,进一步包括,当获取锁发生时,指示该线程将进入或已进入临界区。

13.权利要求11和12中任一项所述的方法,进一步包括,通过功率控制单元,根据进入临界区的线程向核心分配额外的功率预算。

14.权利要求13所述的方法,进一步包括,通过功率控制单元确定多个核心中每个核心的适当的P状态。

15.根据权利要求13所述的方法,进一步包括,通过功率控制单元检测具有已进入节能

状态的线程的多个核心的功率降低。

16.根据权利要求13所述的方法，包括通过功率控制单元增加具有已进入临界部分的线程的核心的电压和频率。

17.权利要求11-16中任一项所述的方法，进一步包括，通过已进入节能状态的一个或更多个其它线程监测锁是否已释放。

18.根据权利要求17所述的方法，进一步包括，通过一个或更多个其它线程，基于对包括获得锁线程的核心的存储器位置的一个或更多个观察，监控锁是否已被释放。

19.根据权利要求17所述的方法，进一步包括，通过一个或更多个其它线程确定锁是否已释放，由此，一个或更多个其它线程的至少一个线程试图获取共享数据的锁。

20.权利要求11-19中任一项所述的方法，进一步包括为节能状态分配选定的C状态。

21.一种用于管理访问用户空间共享数据的方法，包括：

通过处理单元的核心，确定在核心中的线程获取了锁，所述锁指示用户空间的共享数据的处理；

通过所述核心，生成所述核心已获得锁的通知，其中所述通知指示一个或更多个线程尝试访问所述用户空间的共享数据以进入节能状态。

22.根据权利要求21所述的方法，进一步包括：

通过功率控制单元，基于进入临界区的线程为所述核心分配额外功率预算。

使用C状态和睿频加速提高用户空间的自旋锁效率

技术领域

[0001] 本应用程序涉及通过使用内核结构的多线程有效地保护同时访问用户空间共享数据的系统和方法。

背景技术

[0002] 随着当今芯片多处理器中的CPU核心数量在不断增长,但应用程序正变得越来越多线程。尽管多线程应用程序中的线程旨在独立处理其各自的任务,但它们仍然共享一定数量的数据。共享数据访问需要使用同步原语进行保护;否则,如果同时写入,这些数据可能会处于不一致的状态。

[0003] 自旋锁是一种保护共享数据不被多线程同时访问的内核结构原语。在操作中,线程检查用于在共享数据上锁定线程操作的临界区(Critical Section)的锁变量是否可用。当锁变量被启用,它保护共享数据不会被多个线程同时获取以执行其任务。这是至关重要的,因为如果允许多个线程访问同一共享数据,共享数据将变得不稳定。如果锁变量是自由的,即,没有被其它线程使用,则寻找锁变量可用性的线程可以在进入临界区之前获取它。另一方面,如果锁定变量不是自由的,例如,当锁变量被另一个线程获取时,寻找获取锁变量的线程在锁上“自旋”直到它可用。例如,线程等待轮到它。

[0004] 因为自旋锁避免操作系统进程重新调度或上下文切换的开销,如果线程仅在短时间内被阻塞,则自旋锁是高效的。然而,如果自旋锁保持更长的时间,就会变得浪费,因为它们可能会防止其它线程运行并需要重新调度。线程保持有锁的时间越长,线程在保持有锁的同时被操作系统调度器中断的风险就越大。如果发生这种情况,其它线程将保持“自旋”(反复尝试获得锁),而持有锁的线程没有在释放它方面取得进展。结果是无限期推迟,直到持有锁的线程可以完成并释放它。这在单处理器系统中尤其如此,在该系统中,每个具有相同优先级的等待线程可能会浪费其时间量(线程可以运行的分配时间)自旋直到保持锁的线程最终完成。

[0005] 该问题在当前的多处理器中也能看到,在多处理器中,CPU核心数量不断增长,应用程序变得越来越多线程。尽管多线程应用程序中的线程旨在独立处理其各自的任务,但仍有一定数量的共享数据。共享数据访问需要使用自旋锁或类似的方式进行保护,否则如果同时被写入,共享数据可能处于不一致的状态。即使当前的应用程序是多线程的,从所有线程访问临界区仍然是序列化的工作,这已经放大了“繁忙等待”时间。

[0006] 如上所示,传统的自旋锁可能不利于系统的吞吐量。如果系统运行多个任务,一个任务中的线程可能不必要地占用CPU,而没有任何进展。例如,常规自旋锁的替代方案是互斥锁。不能获取锁的线程不会占用CPU继续重试获取锁,而是将CPU交给其它任务。在消除没有产生有用工作的周期的同时,互斥锁对让出CPU的线程有着显著的性能开销。这是因为让出CPU以及重新安排重新获得锁,需要调用操作系统调度器来执行高代价的上下文切换。此外,互斥锁是仅在操作系统内核中可用的同步原语。由于它需要对操作系统调度器进行主动调用,因此不能在用户空间中使用互斥锁。

发明内容

[0007] 本公开的实施例提供一种处理系统和方法，所述系统和方法通过使用例如改进的用户空间自旋锁的内核结构的多线程有效地保护对用户空间共享数据的同时访问。

[0008] 本公开的实施例还提供一种处理系统和方法，该系统和方法通过存储器访问用户空间的共享数据给应用程序的多个线程，并由多个核心执行多个线程中的一个或更多个线程，其中，多个核心的其中一核心被配置为通过线程获取指示共享数据的处理的锁，并生成核心已获取锁的通知，其中，所述通知指示试图访问共享数据的一个或更多个其它线程进入节能状态，其中所述节能状态是选择的C状态。

[0009] 本公开的实施例还通过获取锁提供线程将进入或已经进入临界区的指示。该处理系统和方法进一步包括电源控制单元，该电源控制单元被配置为基于进入临界区的线程向核心分配额外的功率。电源控制单元进一步被配置为多个核心中每个核心的确定适当的P状态，并且检测具有进入节能状态的线程的多个核心的功率减少。电源控制单元进一步被配置为增加具有已进入临界区的线程的核心的电压和频率。

[0010] 本公开的实施例还通过已进入节能状态的一个或更多个其它线程提供是否已释放锁的监控，其中，所述监控基于对包括已获得锁的线程的核心的存储器位置的一个或更多个的观察，并通过一个或更多个其它线程确定该锁已被释放，并试图通过一个或更多个其它线程的至少一个线程获取对共享数据的锁。

[0011] 所公开的实施例的附加目的和优点将在以下描述中部分阐述，并且部分将从描述中显而易见，或者可以通过实施例的实践来学习。所公开的实施例的目标和优点可以通过权利要求中规定的要素和组合来实现和获得。

[0012] 应当理解的是，前述的一般描述和以下详细描述仅是示例性和解释性的，并不限制所公开的实施例。

附图说明

[0013] 图1是使用示例性x86汇编语言伪代码来实现常规的自旋锁的框图。

[0014] 图2是示出与常规自旋锁相关的示例性性能开销的框图。

[0015] 图3是与本公开的实施例一致的示例性处理系统的示意图。

[0016] 图4是示出与本公开的实施例一致的用户空间中改进的自旋锁的示例性工作机制的框图。

[0017] 图5是表示与本公开的实施例一致的与用户空间中改进的自旋锁相关联的性能开销的示例性方法的流程图。

[0018] 图6是与本公开的实施例一致的使用示例性x86汇编语言伪代码在用户空间中实现改进的自旋锁的框图。

具体实施例

[0019] 现在将详细参考示例性实施例，其示例在附图中进行了说明。以下描述所参考的附图中，除非另有说明，不同附图中的相同数字代表相同或相似的元素。以下示例性实施例中的描述所阐述的实施方式并不代表与本发明一致的所有实施方式。相反，它们仅仅是与附加权利要求中所述的与本发明相关的方面相一致的设备和方法的例子。

[0020] 所公开的实施例提供了一种在访问用户空间中的临界区时可以实现高性能的改进的自旋锁。改进的用户空间自旋锁可用于高速固件以实现低延迟和高带宽。例如，今天的高性能服务器系统通常配备提供高带宽和低延迟I/O的高速固态驱动器(SSD)和智能网卡(SmartNIC)。所公开的实施例可以使用专用的用户空间线程来替换操作系统内核代码来访问这些高速设备，例如数据平面开发套件存储(Data Plane Development Kit,DPDK)和存储性能开发套件(Storage Performance Development Kit,SPDK)。此外，改进的用户空间自旋锁对具有扩展共享数据访问的多线程应用程序的性能也至关重要，例如关系数据库管理系统(Relational Database Management System,RMDBS)。

[0021] 常规的自旋锁是保护共享数据不被多线程同时访问的内核结构原语。现在请参考图1，其为使用示例性的x86汇编语言伪代码来实现常规自旋锁的框图。线程主体中访问共享数据的代码称为临界区。每个临界区通常都使用锁变量进行保护，例如lock_var，其在任何给定时间专门授予一个线程。因此，线程需要在进入临界区之前竞争获得锁。无法获取锁的线程将继续尝试获取锁，直到成功。换句话说，锁的持有者在完成执行其临界区后释放锁。由于线程保持活动状态，但却没有执行有用的任务，使用这样的锁忙于等待。在忙于等待期间，线程没有产生有用的工作，却完全占用CPU并消耗了大量的能量。自旋锁一旦被获得通常会被持有，直到被明确释放，尽管在一些等待的线程(持有锁的线程)阻塞或进入睡眠状态时可能会被自动释放。

[0022] 返回图1，执行获取锁的实际指令(第3行中的xch)是由底层CPU指令集提供的原子指令。原子指令确保指令执行的原子性，它要么完成整个指令包，要么失败。原子指令通常用于锁定访问的整个缓存和存储器总线，并拒绝从其中读取或写入任何其它内容。这使得原子指令的执行代价很大。在图1所示的代码片段中。用于自旋的代码依赖于原子指令，这使得自旋的性能无法接受。

[0023] 现在参考图2，其为示出与常规自旋锁相关联的示例性性能开销的框图。图2中的性能开销是说明性的，因为锁的获取按顺序显示为由线程T_0-T_N获取。然而，在操作中，所述顺序获取并非总是如此。线程T_0-T_N可以以任何随机顺序获取锁。在N个线程竞争锁的情况下，可以保证N-1个线程在第一轮将竞争失败。因此，它们都必须等待已成功获取锁的线程。当持有锁的线程完成其临界区并释放锁时，其它N-1个线程将再次竞争锁，N-2个线程将会失败，从而在锁上自旋。结果，自旋的量总是临界区长度的$O(N^2)$，其中，O代表当参数趋向于一个特定的值或无穷大时，函数的复杂度或极限行为的顺序。

[0024] 在操作中，用于自旋的代码通常使用行业标准来实现，例如依靠常规读取指令来获取锁变量的测试和设置(test and set)，或者首先使用常规读取指令首先获取锁变量，并在重新发出读取之前在while循环上自旋一段时间的测试和测试和设置(test and test and set)。然而，即使如此优化，CPU仍然完全被自旋的线程占据，在消耗能量的同时没有产生有用的工作量。此外，因为自旋的线程需要不断地对存储在存储器中的锁变量进行读取，它们可能会干扰在临界区运行的线程的执行。

[0025] 返回至图2，线程尝试在A_1获取锁，并在B_1成功获取。与此同时，线程T_1-T_N继续在A_2-A_{N+1}尝试获得锁。由于线程T_0在B_1成功获得锁，线程T_1T_N在E_1-E_N自旋，即等待线程T_0完成其任务。线程T_1-T_N自旋，直到线程T_0在临界区C_1。当线程T_0完成其任务并在D_1释放锁时，线程T_1-T_N重新尝试以获取锁；线程T_1在此重试时在B_2成功获得锁，但线程T_2(未显示)-T_N在E_2-E_N继续

自旋(未显示),直到线程T_1处于临界区C_2。

[0026] 当线程T_1在D_2释放锁时,线程T_2(未显示)-T_N重试获取锁,并继续该过程,直到线程T_N在B_{N+1}获得锁,在C_{N+1}进入临界区,并在D_{N+1}释放锁。从本视图可知,线程T_N获取锁、进入临界区和释放锁所需的总时间,已大大超过了线程T_0获取锁、进入临界区以及释放锁所需的时间。因此,自旋的量总是临界区长度的$O(N^2)$。由于线程T_N自旋持续最长的时间(E_N的长度超过了任何先前的自旋框的长度),总吞吐量直接取决于获取锁的线程数以及每个线程必须花费在自旋的时间。

[0027] 图3是与本公开的实施例一致的示例性处理系统300的示意图。处理系统300可以包含在服务提供商的基于云的服务器中。用户设备390可以通过网络访问服务器。如图3所示,处理系统300包括处理单元310、高速缓存350、系统内核370和连接到处理单元310的主存储器380。主存储器380可以存储通过处理单元310访问的数据。系统内核370可以控制处理系统300的操作。处理系统300包括系统内核370和存储单元372,所述存储单元372存储描述要在处理系统300上执行的一个或更多个任务/线程的属性的task_struct数据结构。

[0028] 处理器310和高速缓存350可以包含在CPU芯片中,其中处理器310被设置在CPU裸片(die)上,而高速缓存350被设置于在物理上与CPU裸片相分离的裸片上。处理单元310包括多个处理核心322a-d,分别对应于并连接到多个处理芯片322a-d以及连接到框架(fabric)326的多个二级高速缓存(Level-2 cache,L2C)324a-d。此外,处理单元310包括电源控制单元(PCU)328、最后一级高速缓存(LLC)330(可选择)和控制电路340。高速缓存350包括高速缓存数据阵列352。

[0029] PCU 328在其固件中运行电源算法,以确定每个内核322a-d的适当P状态。P状态为处理单元310中的每个功率岛都有预定义的频率和电压点。一般来说,更高的电压将与更高的频率相关联,从而导致高功耗。

[0030] 现在的CPU通常定义几CPU电源状态,也被称为C状态,例如在Intel®'的x86中定义的C_0-C_6。当CPU核心在正常运行时,它处于C_0状态,所有CPU资源都在运行可使用。当它进入更深的C状态时,它的部分资源要么是时钟门控的,要么是功率门控的。例如,在C_1状态,CPU核心的时钟被门控,导致核心处于暂停状态,而L2高速缓存仍然完全运行。

[0031] 当时钟被门控时,被时钟门控的部分的输入时钟停止,因此,不会发生逻辑切换,从而节省动态功率。当功率被门控时,输入到功率所驻留的功率岛的所述功率被关闭,从而使整个部分处于电源关闭状态,并节省动态和静态功率。功率门控基本上失去了存储在CPU部分的当前状态,并且需要一些关键状态保存在保留触发器中,或者在关闭电源之前刷新到存储器中。

[0032] 时钟门控(例如x86中的C_1状态)的性能影响可以忽略不计,因为停止和授予时钟几乎没有延迟。与功率门控相比,时钟门控并没有节省太多功耗。这并没有给在临界区睿频加速核心留下太多余地。另一方面,功率门控的性能影响(例如在x86的C_2状态-C_7状态)是重大的。平均而言,在最新的x86 CPU中从C_2状态过度到正常C_0状态的延迟大约为1微秒(μs),从C_6状态返回到C_0状态可能是几十微秒(μs)。

[0033] 本公开的实施例还在改进的用户空间自旋锁(或改进的自旋锁)中提供减少与自旋锁实施相关的性能和功耗干扰的机制。本公开的实施例还提供了临界区中线程通过增加运行该线程的CPU核心的频率和电压来更快完成其执行的能力。

[0034] 根据用户空间中改进的自旋锁的实施例，所述改进的自旋锁也还可以提供作为库函数。特别是，提供了多个这样的API库，其中的每个API都允许进入一特定的C状态，例如spinlock_C_1、spinlock_C_2等。实际上，程序员可以能够选择使用哪个用户空间。根据进一步的实施例，例如，具有更深C状态的用户空间自旋锁API使用较长的临界区。这是因为冗长的临界区可以轻松的摊销C状态转换回C_2-State的延迟。

[0035] 根据实施例，提供了一种在改进的自旋锁中减少与自旋锁实施相关的性能和功率干扰的机制。根据进一步的实施例，增加CPU核心的频率和电压，为临界区的线程提供更快完成其执行的能力。特别的，实施例利用CPU提供的C状态和睿频加速技术。根据进一步的实施例，在CPU资源的时钟门控期间提供了动态功率的节省。根据进一步的实施例，在CPU资源的功率门控期间提供了动态和静态功率的节省。

[0036] 本公开的实施例还提供改进的自旋锁中的新指令，以允许线程在用户空间进入节能状态，例如图4中的线程T_1；并允许临界区的线程4指示CPU中的功率控制单元(Power Control Unit，PCU)专门为执行指令的核心分配净空功率预算，例如图4中的线程T_0。

[0037] 本公开的实施例还提供在CPU资源的时钟门控期间供动态功率节省。本公开的实施例还提供在CPU资源的功率门控期间动态功率和静态功率节省。本公开的实施例也提供了改进的自旋锁作为库函数。

[0038] 现在请参考图4，其为与本公开的实施例一致的示出在用户空间中的改进的自旋锁的示例性工作机制的框图。临界区仍然使用锁变量进行保护，并且线程需要在进入临界区之前竞争来获得锁，以便访问共享数据。在未能获得锁之后，所有剩余的线程进入低功耗CPU状态以节省电力，而不是在锁上自旋。进入低功耗状态的多个CPU核心根本上降低了整个CPU封装的有效功耗，这反过来为当前运行的核心通过增加运行核心的输入电压进入更高的P态(或睿频加速)创造了空间。结果，临界区的线程可能会更快完成。在线程完成其工作并即将离开临界区之前，它负责唤醒当前处于节能状态的其它线程。一旦被唤醒，这些线程将继续竞争获取锁。

[0039] 返回图4，线程T_0试图在A_1获取锁，并在B_1成功获取到。同时，线程T_1-T_N也试图在A_2-A_{N+1}获得锁。由于线程T_0在B_1已成功获得锁，线程T_1-T_N进入节能状态P_1-P_N，等待线程T_0完成其的任务。在节能状态P_1-P_N期间，PCU检测到CPU核心由于线程获取锁失败而降低功耗，这些线程处于更深的C状态。现在整个CPU封装功率中都有可用的余量。这允许PCU增加正在运行的CPU核心的电压和频率。因此，在临界区运行的线程(T_1)可以更快地完成。

[0040] 返回图4，线程T_1-T_N保持在节能状态P_1-P_N直到线程T_0到在临界区C1。当线程T_0完成其任务时，线程在D_1释放锁之前唤醒其它等待线程T_1-T_N。线程T_1-T_N在R_1-R_N分别重试以获取锁。在此重试时，线程T_1在B_2成功获取，但是线程T_2(未显示)-T_N继续保持在节能状态，P_2(未显示)-P_N直到线程T_1在临界区C_2。当线程T_1完成其任务时，线程T_1在D_2释放锁之前唤醒其它线程T_2(未显示)-T_N。线程T_2(未显示)-T_N继续该过程，直到线程T_N在B_{N+1}处获得锁，在C_{N+1}进入临界区并在D_{N+1}释放锁。从此处示图中可知，线程T_N获得锁、进入临界区和释放锁所需的总时间，比如图2所示的线程T_N获得锁、进入临界区和释放锁所需的总时间要少得多。

[0041] 现在请参考图5，其为与本实施例一致的表示与用户空间中改进的自旋锁相关的性能开销的示例性方法500的流程图。请参考图5，很容易理解，如图所示的程序可以做如下所示被修改以删除步骤或进一步包括额外的步骤。此外，步骤可以以不同于方法500所示的

顺序执行，和/或并行执行。在表示方法500的流程图中，提供了可供处理器(例如，x86 Intel®处理器)在用户空间中实现改进的自旋锁的示例性步骤，应当理解的是，来自其它制造商的一个或更多个其它处理器可以单独或组合在客户端设备(例如,笔记本电脑或蜂窝设备)或后端服务器上执行基本相似的步骤。

[0042] 在初始的开始步骤505之后，一个或更多个线程(例如，图4中的线程T_1-T_N)在步骤510竞争锁。接着，在步骤515的第一线程(例如，图4中的线程T_0)获得锁(例如，在图中4的框B_1)。正如所讨论的，获得锁的线程(例如，在图中4的框B1)，在它执行任务之前进入临界区，例如在步骤520(和在图中4的框C_1)。接着，在步骤525中，检查临界区中的线程(例如，第一线程T_0)是否已经完成其任务。如果第一线程仍然在临界区(步骤525中的“否”分支)，在步骤530，其它等待线程进入节能状态(例如，在图4中的框P_1-P_N。)，流程继续回到步骤520。

[0043] 另一方面，如果第一线程已经完成了它的任务(例如，从步骤525的“是”分支)，在步骤535第一线程在步骤540(例如，在图4中的框D_1)释放锁之前唤醒其它等待的线程(例如，在图4中的框C_1)以便其它线程中的一个可以获得锁(例如，在图4中框A_2-A_N)。接着，在步骤545再次检查是否有其它等待线程，如果有其它等待线程(步骤545中的“是”分支)，在步骤550其它线程竞争锁。接着，在步骤555，第二线程(例如，图4中的线程T_1)获得锁(例如，在图4中的框B_2)并在步骤560进入临界区(例如，在图4中的C_2)。

[0044] 接下来，在步骤565中，再一次检查临界区的线程是否已经完成了任务，例如T_1。如果第二线程仍然在临界区(从步骤565“否”分支)，在步骤580，其它等待线程继续保持节能状态。在图4的框P_2(未显示)-P_N。流程继续回到步骤560。另一方面，如果第二线程已经完成了它的任务(从步骤565的“是”分支)，在步骤570，第二线程在步骤575(例如，在图4的框D_2)的释放锁之前唤醒其它等待线程(例如，在图4的框C_2)，以便其它线程中的一个可以获取锁(例如，在图4的框A_3(未显示)-A_N)，流程回到步骤545。如果在步骤545中不再有等待线程，则该方法在步骤585结束。

[0045] 现在参考图6，其为与本公开的实施例一致使用示例性的x86汇编语言伪代码在用户空间中实现改进的自旋锁的框图。根据实施例，至少包括两个新指令，例如允许线程在用户空间中进入节能状态的umwait指令，和允许临界区的线程指示CPU中的PCU专门为执行pcuhint指令的核心分配净空功率预算的pcuhint指令。

[0046] 返回至图6，线程仍然使用原子指令来获取锁。一旦失败，例如，在第6行，线程首先执行监视器指令来设置要监视的存储器位置，然后执行umwait指令，例如在第7行，输入选定的C状态。应当注意的是，在常规的CPU中，只有通过特权指令在操作系统内核中才能进入节能状态。为了允许用户空间中的常规自旋锁进入C状态，需要新的umwait指令。在操作中，umwait指令的操作与现有的mwait指令类似，因为它接受存储在累加器和计数器中的参数，例如，在x86中的EAX和ECX寄存器以确定所需的要进入的C状态，并将返回的错误代码存储在ECX中。但是现有的mwait指令和新的umwait指令之间的区别到此结束。新的umwait指令允许线程在CPU处于无特权模式被执行，例如英特尔8的ring3，因此在ring 3执行时不会引起一般性保护错误(GP)。

[0047] 在操作中，当执行umwait指令时，未能获得锁的线程停止执行任何指令并进入所需的C状态。这可以防止未获得锁线程的消耗功率，消除对临界区运行核心的功率干扰。它还可以防止未获得锁线程对锁变量进行读取，消除对临界区中运行核心的性能干扰。

[0048] 返回图6,一新的pcuhint指令,例如在第10行,允许在临界区运行的线程与PCU通信。一旦执行,PCU将把所有剩余的功率预算分配给已执行pcuhint的核心,从而提高核心的效率。若没有pcuhint,就像在常规的自旋锁中一样,PCU可以对所有当前运行的内核进行同等的功率预算。由于可能有运行不相关任务的核心,运行在临界区的核心可能会在没有pcuhint的情况下获得较小的功率提升。值得注意的是,pcuhint不需要操作数,因此可以在无特权模式下执行。

[0049] 在临界区的线程即将离开临界区之前,它还执行存储指令,例如,存储到在第11行的在C状态的线程监视的存储器位置。相应地,这些线程将被唤醒,转换回C0状态,并接受下一个指令,例如在第8行的进入C-State之前以恢复执行。因此,这些线程将跳回图6所示代码段的开头重试锁。

[0050] 在前述详细说明中的实施例已经参考了许多具体细节,这些细节可以从实施到实施而有所不同。可以对所描述的实施例进行某些调整和修改。考虑到本发明公开的具体说明和实践,对于本领域的技术人员来说其它实施例是显而易见的。应当理解本公开的具体说明和实例仅仅是示例性的,本发明的真实范围和精神由文本的权利要求所表示。应当理解在附图中所示出的步骤顺序也仅用于说明目的,其无意限定于任何特定的步骤顺序。因此,那些技术人员可以理解,这些步骤可以在执行相同方法时以不同的顺序执行。

```
1. spin_lock:
2.     mov  eax,  1
3.     xch  eax  [lock_var]  ;     获取锁
4.     text  eax,  eax
5.     jnz   spin_lock          ;获取锁失败，再试
6.                              ;获取锁成功
7. Enter Critical Section
                ...
8. Leave Critical Section
9. spin_unlock:
10.    mov  0,  [lock_var]          ;     释放锁
```

图1

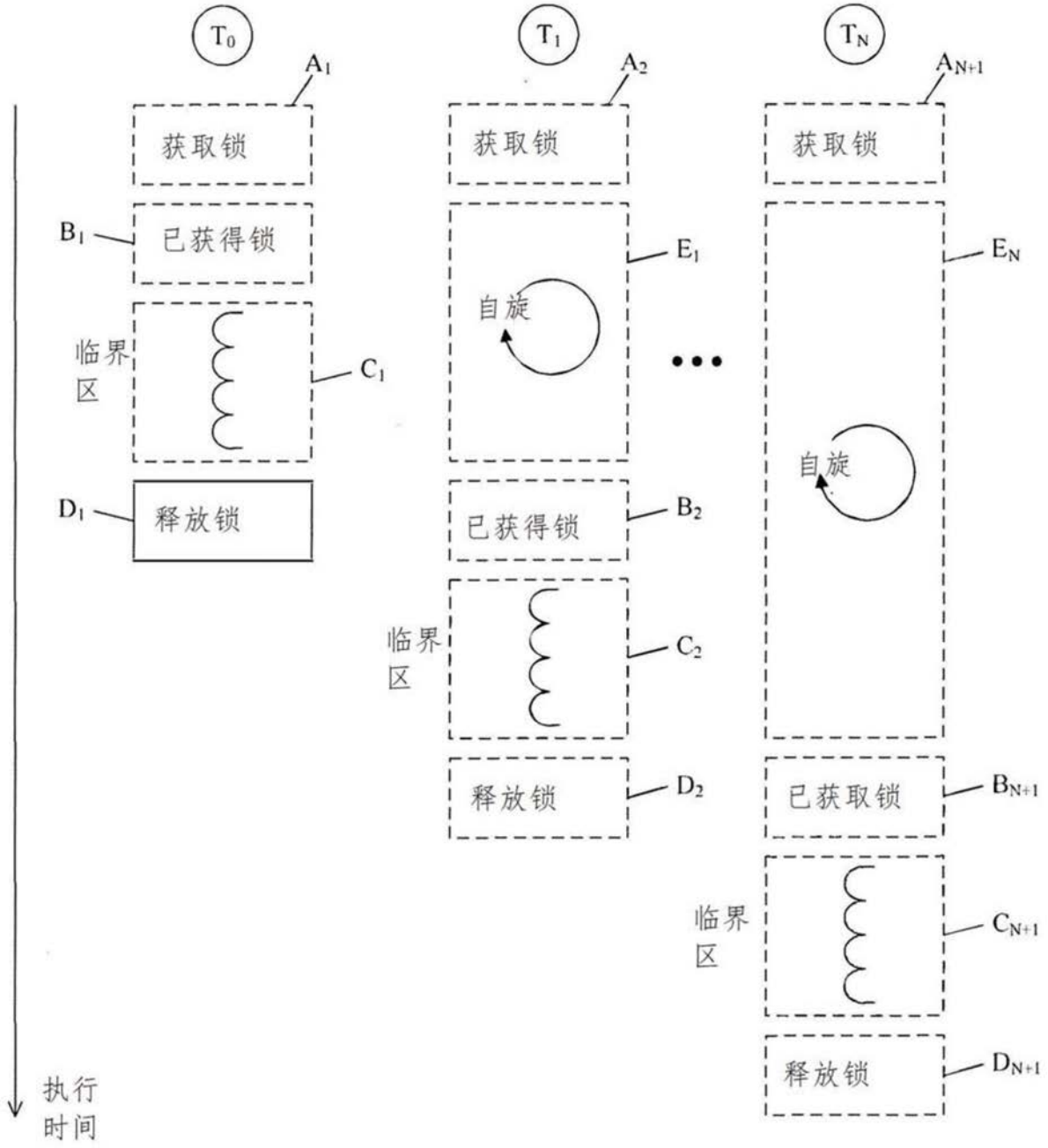

T0
A1
获取锁
B1
已获得锁
临界区
C1
D1
释放锁
T1
A2
获取锁
E1
自旋
B2
已获得锁
临界区
C2
释放锁
D2
TN
AN+1
获取锁
EN
自旋
已获取锁
BN+1
临界区
CN+1
释放锁
DN+1
执行时间

图2

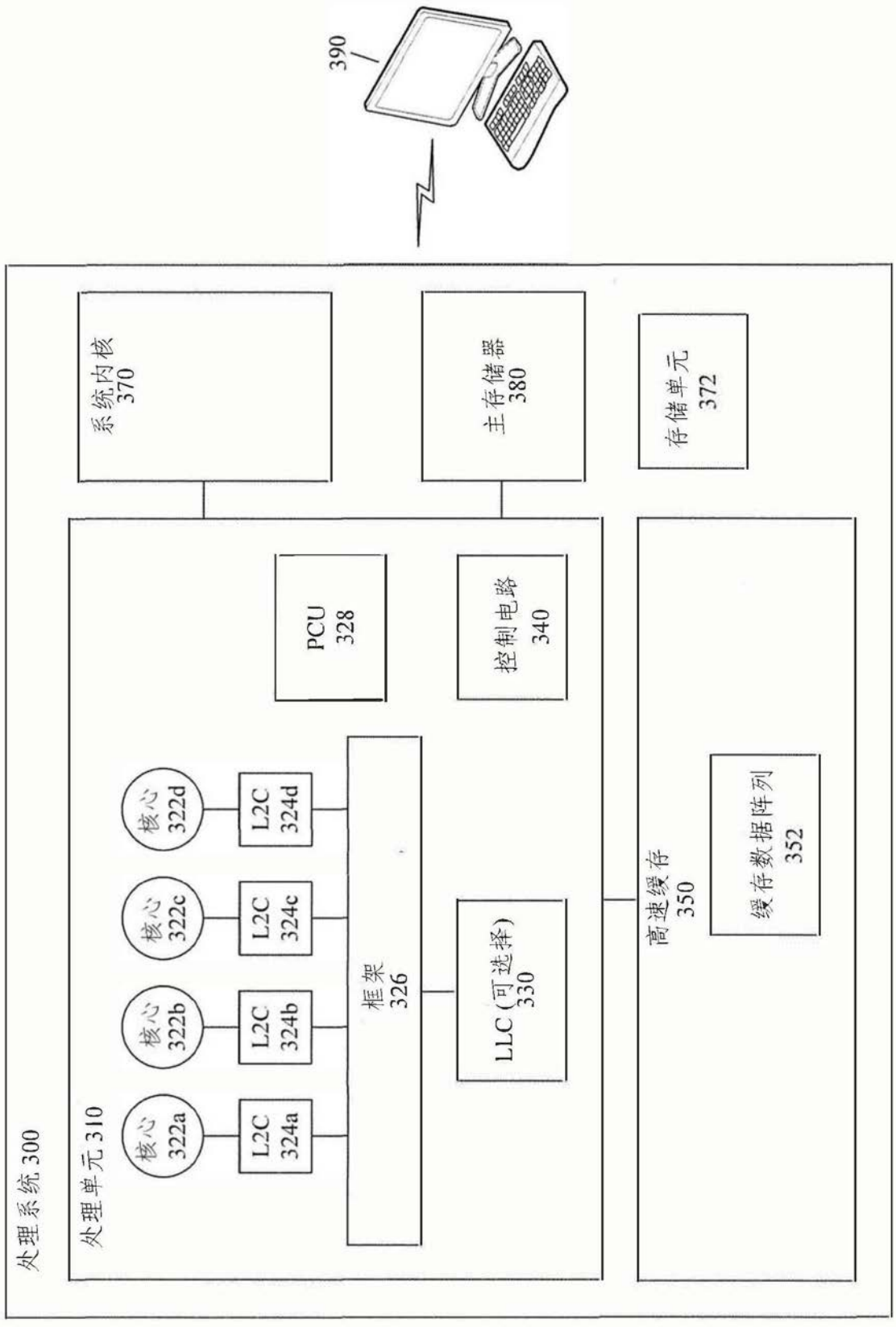

390
处理系统 300
处理单元 310
核心 322a
核心 322b
核心 322c
核心 322d
L2C 324a
L2C 324b
L2C 324c
L2C 324d
框架 326
LLC (可选择) 330
PCU 328
控制电路 340
系统内核 370
主存储器 380
存储单元 372
高速缓存 350
缓存数据阵列 352

图3

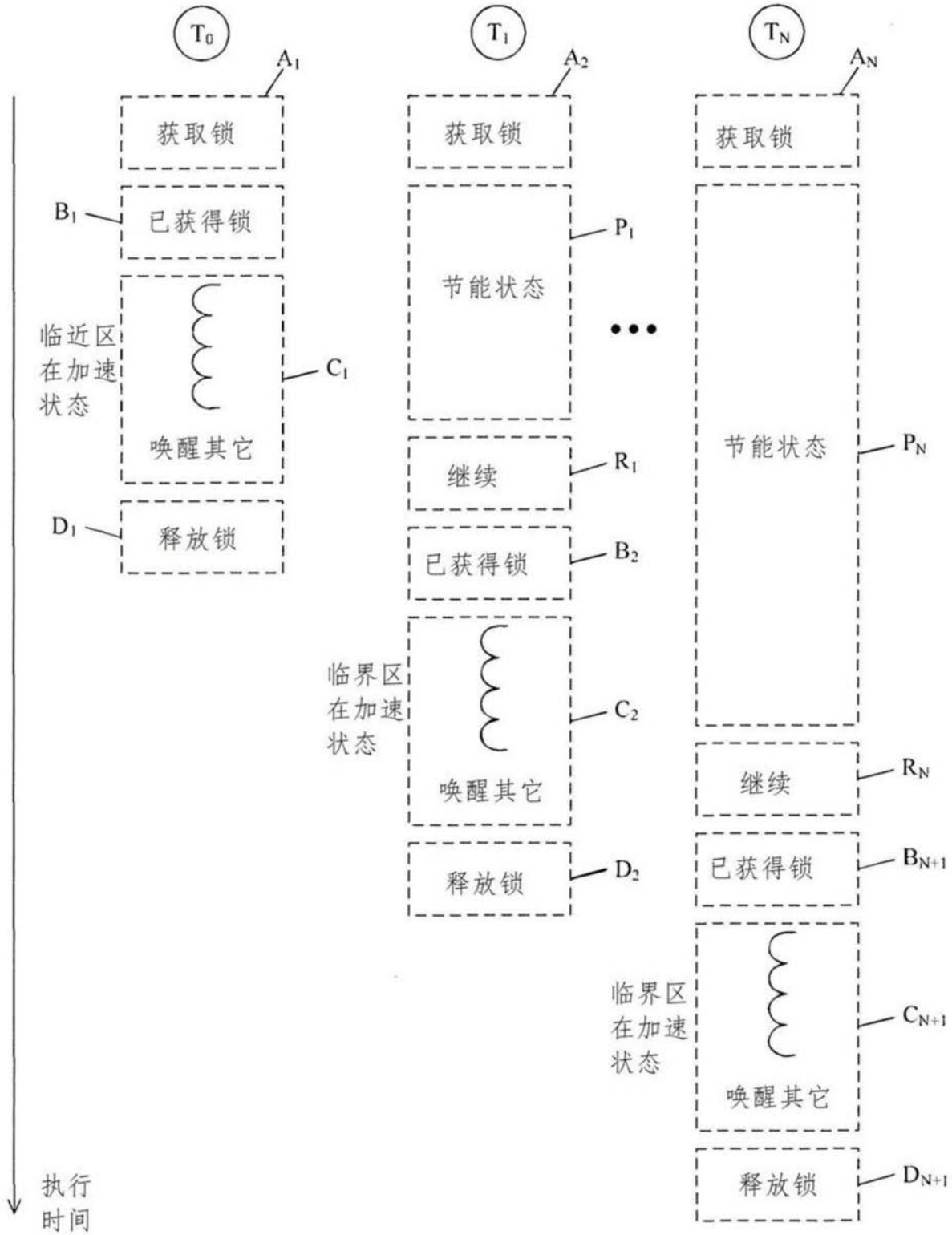

T0
T1
TN
A1
A2
AN
获取锁
已获得锁
B1
临近区在加速状态
C1
唤醒其它
D1
释放锁
节能状态
P1
PN
继续
R1
RN
B2
临界区在加速状态
C2
D2
BN+1
CN+1
DN+1
执行时间

图4

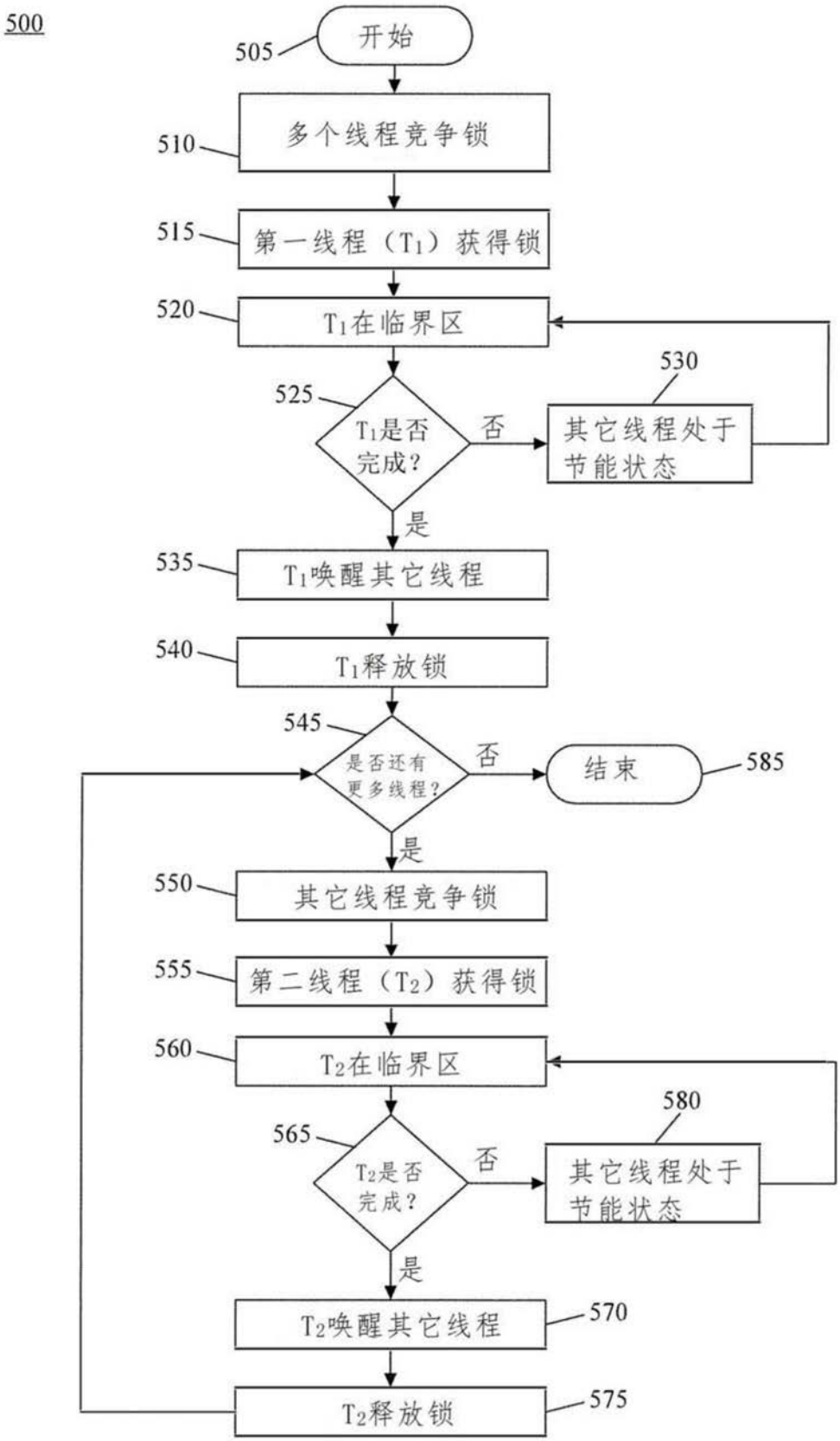

500
505
开始
510
多个线程竞争锁
515
第一线程（T1）获得锁
520
T1在临界区
525
T1是否完成？
否
530
其它线程处于节能状态
是
535
T1唤醒其它线程
540
T1释放锁
545
是否还有更多线程？
否
结束
585
是
550
其它线程竞争锁
555
第二线程（T2）获得锁
560
T2在临界区
565
T2是否完成？
否
580
其它线程处于节能状态
是
570
T2唤醒其它线程
575
T2释放锁

图5

```
1. spin_lock_Cx:
2.      mov         eax,   1
3.      xch         eax    [lock_var]   ; 获取锁
4.      text        eax,   eax
5.      jz          critical_section    ; 获取锁成功
6.      monitor     mem_location
7.      unwait      Cx_State            ; 获取锁失败，进入低功率状
                                        ; 态
8.      jmp         spin_lock_Cx        ; 唤醒并重试
9. critical_section:
10.     pcuhint                         ; 指示PCU并使其进入加速状态
              ...                       ; 临界区在加速状态
11.     mov 1,      [mem_location]      ; 唤醒其它
12. Leave Critical Section
13. spin_unlock:
```

图6