

****

Topic 4

Synchronization

|  |  |  |
| --- | --- | --- |
| 学院名称 | 计算机学院 | |
| 林新源ZY2006352 | | 同步硬件原语(25%) | |
| 温雅楠SY2006345 | | 同步软件原语(25%) | |
| 武仕沛ZY2006357 | | 自旋锁(25%) | |
| 王郁含ZY2039124 | | 栅障(25%) | |

2021年 1 月30日

目录

[1 绪论 1](#_Toc62984026)

[2 同步硬件原语：Test\_and\_Set 2](#_Toc62984027)

[2.1 基本硬件原语 2](#_Toc62984028)

[2.2 Test\_and\_Set，Unset，QOSB 2](#_Toc62984029)

[2.2.1 Test\_and\_Set，Unset，QOSB的语义 2](#_Toc62984030)

[2.2.2 Test\_and\_Set，Unset，QOSB的应用 2](#_Toc62984031)

[2.2.3 基于缓存一致性的Test\_and\_Set，Unset，QOSB的实现 3](#_Toc62984032)

[2.3 LL/SC指令对 4](#_Toc62984033)

[2.3.1 复杂性问题 4](#_Toc62984034)

[2.3.2 原子性表象的实现——LL/SC指令对 4](#_Toc62984035)

[2.3.3 利用LL/SC指令对实现锁 4](#_Toc62984036)

[3 同步软件原语：Fetch\_and\_φ 6](#_Toc62984037)

[3.1 热点问题及其性能影响 6](#_Toc62984038)

[3.1.1 热点问题与fetch\_and\_φ的关系 6](#_Toc62984039)

[3.1.2 树饱和 6](#_Toc62984040)

[3.2 硬件fetch\_and\_φ原语实现——消息组合技术 6](#_Toc62984041)

[3.3 硬件fetch\_and\_φ原语实现——软件组合树 7](#_Toc62984042)

[3.3.1简单的软件组合树 7](#_Toc62984043)

[3.3.2 二叉组合树 7](#_Toc62984044)

[4 自旋锁Spin lock 11](#_Toc62984045)

[4.1 自旋锁简介及性能指标 11](#_Toc62984046)

[4.2 各类锁 12](#_Toc62984047)

[4.2.1 TS锁 12](#_Toc62984048)

[4.2.2 TTSL锁 14](#_Toc62984049)

[4.2.3 Ticket锁 16](#_Toc62984050)

[4.2.4 ABQL锁 18](#_Toc62984051)

[4.3 总结 19](#_Toc62984052)

[5 栅障Barrier 20](#_Toc62984053)

[5.1 栅障的目的 20](#_Toc62984054)

[5.2 软件栅障 20](#_Toc62984055)

[5.2.1 集中式栅障 20](#_Toc62984056)

[5.2.2 组合树栅障 22](#_Toc62984057)

[5.2.3 蝴蝶栅障 22](#_Toc62984058)

[5.2.4 散播栅障 22](#_Toc62984059)

[5.2.5 比赛栅障 22](#_Toc62984060)

[5.3 理论对比 23](#_Toc62984061)

[5.4 硬件栅障 23](#_Toc62984062)

[结束语 25](#_Toc62984063)

[参考文献 25](#_Toc62984064)

# 1 绪论

在并行多核共享存储的系统中，互相协作的进程或线程通过共享逻辑地址空间来实现共享数据。在这样的情况下，如果没有对共享数据加以保护，那么共享数据的并行访问和读写可能会产生违反直觉的结果，即会产生数据的不一致的情况。产生这种数据不一致的原因在于多个进程或线程在对共享数据进行读写操作时发生了争用即发生了竞争条件，而此时没有相应的机制对它们的读写进行协调，导致对共享数据的读写非常混乱。因此，一个可行的解决方案为，在并行多核共享存储的系统中引入同步机制对共享数据加以保护，使得数据保持顺序一致性。

本文主要讨论并行多核共享存储系统的同步机制的实现问题。第2节将讨论一个同步基本硬件原语——Test\_and\_Set。第3节将讨论如何用软件方法实现fetch and φ原语，第4节将讨论自旋锁及其实现。第5节将讨论栅障及其实现。

# 2 同步硬件原语：Test\_and\_Set

## 2.1 基本硬件原语

在并行多核共享存储系统中实现同步所需要的关键功能就是一组能够以原子方式读取和修改存储器位置的硬件原语。这里的“原子”意味着两点，第一，要么整个序列的指令都被完整执行，要么其中任何一条指令都不执行。第二，在任何给定的时间内，只有一条原子指令能够被执行。大部分处理器是通过支持原子指令来实现同步所需要的硬件原语。有了基本硬件原语，我们就可以实现更多的同步操作，例如锁和栅障。之后的章节会详细介绍锁和栅障。下面，我们将以有关同步基本原语的经典论文——文献[1]为例，介绍在缓存一致性的基础上如何实现一组同步所需的原子指令。

## 2.2 Test\_and\_Set，Unset，QOSB

### 2.2.1 Test\_and\_Set，Unset，QOSB的语义

文献[1]采用一个先进先出的同步位（syncbit）队列的方式对处理器进行排序，每个数据行的同步位创建相应的队列来实现对共享数据的读写的顺序一致性来实现同步。基于同步列队列的方式，文献[1]给出了3种原子指令：Test\_and\_Set，Unset，QOSB。下面介绍它们的语义。

Test\_and\_Set的语义为当处理器处于队列头部（或没有队列时）时，测试一个同步位，若为“unset”，则Test\_and\_Set成功，并置同步位为“set”，返回测试的结果。

Unset的语义为将指定同步位置为“unset”，然后将处理器从队头移除。

QOSB是一种非阻塞操作，其语义为如果一个处理器还没加入处理器所需数据行对应的同步位的队列，QOSB操作将处理器加入该同步位队列，若该同步位没有队列，则为该同步位创建队列。如果处理器未处于队头进行Test\_and\_Set操作，那么此时不进行同步位测试即失败（即返回“set”）。

### 2.2.2 Test\_and\_Set，Unset，QOSB的应用

（I）锁的实现

利用Test\_and\_Set，Unset，QOSB三种原子指令进行锁的实现如图2-1。锁的实现包括申请锁和释放锁过程的实现。申请锁时，首先需要进行QOSB操作将处理器加入对应的同步位队列，然后不断对对应的同步位进行Test\_and\_Set操作直到Test\_and\_Set成功，这期间同步位队列有可能发生损坏，此时需不断进行QOSB操作将未加入同步位队列处理器加入对应的同步位队列。Test\_and\_Set成功时，将同步位由“unset”置为“set”，此时表示申请到锁。由于此时申请到锁的处理器处于同步位队列的队头，并且同步位为“set”，其它处理器要对该同步位申请锁时Test\_and\_Set操作会失败。释放锁时，只需要对同步位进行Unset操作将同步位置为“unset”，并将处理器从同步位队列移出即可。

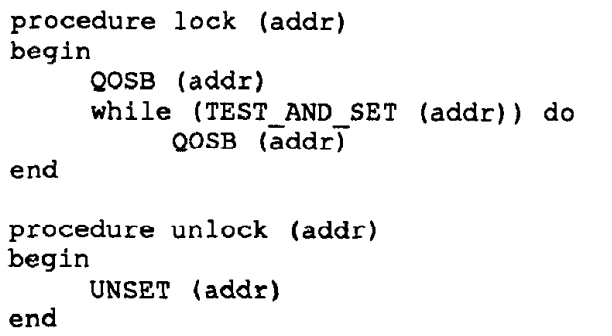


图2-1 利用三种原子指令实现锁

（II）预取的实现

我们还可以利用QOSB来实现预取操作，其实现方法如图2-2。一般而言，如果一个程序执行中访问了一个数组中的一个元素，那么其相邻元素在接下来的程序执行中很有可能被访问到，由此我们可以对这些相邻元素进行QOSB操作将处理器提前加入它们的同步位队列。进而减少因同步实现引入的延迟。

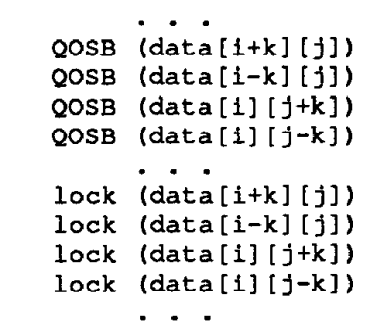


图2-2 利用三种原子指令实现预取

### 2.2.3 基于缓存一致性的Test\_and\_Set，Unset，QOSB的实现

Test\_and\_Set，Unset，QOSB三种原子指令的硬件实现主要是在缓存一致性协议的基础上增加缓存的状态来实现三种原子指令。对于Test\_and\_Set和Unset操作，在MSI缓存一致性协议的基础上增加了状态Locked。Locked和MSI缓存一致性协议中的M状态类似，只有本缓存有唯一的拷贝，并且即将被写入，不用通过总线传输。和M不一样之处当缓存数据行的状态为Locked时，其同步位的状态为set。当进行Test\_and\_Set操作时，系统会原子性地读取指定行同步位的值，并且set。如果数据在本地存在，缓存状态变为Locked。如果此时缓存处于S状态，则改为M状态。在进行Test\_and\_Set操作时，如果同步位处于set状态，返回失败信息，否则先将缓存状态修改为M状态，在得到数据行后，将缓存状态置Locked。在进行Unset操作时，缓存一致性协议将Unset操作视为一个写操作。即将缓存状态变为M状态，将其它对应的缓存状态置为I状态。

QOSB有两种重要的操作，一是在本地缓存分配空间给一个伴随着影子同步位（shadow syncbit）为set的影子拷贝（shadow copy），二是执行一个远程访问，获得数据行的专有拷贝。在MSI缓存一致性协议的基础上，增加了两个缓存状态：Shadow状态和Sticky状态。Shadow状态表示QOSB操作已经发生，此时处理器在队列中但不在同步位队列的队头，并且数据行是无效的，同步位状态为set。Sticky状态表示此时数据行是有效的，同步位为unset，并且此时处理器在同步位队列的队头。由于影子拷贝的数据是无效的，因此我们可以利用影子拷贝的空间来存储队列的指针。

## 2.3 LL/SC指令对

### 2.3.1 复杂性问题

2.2节对原子指令的实现是采用了为原子指令的整个持续时间保留一个高速缓存块的方法，实现起来代价比较高昂。这种局限性存在的主要原因在于支持原子指令实现一致性会带来复杂性。为了避免支持原子指令带来的复杂性问题，可以采用提供原子性表象的方法，即执行结果实现原子性，来保证顺序一致性。

### 2.3.2 原子性表象的实现——LL/SC指令对

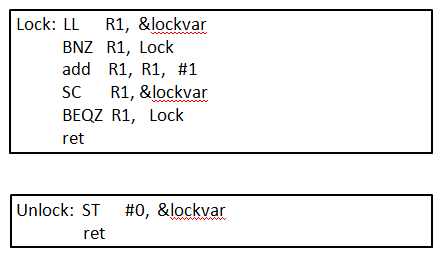
实现原子性表象的一种方法是使用一对指令，其中第二条指令可以返回一个值，通过这个值来判断这一对指令的执行是否是原子性地执行。如果任意处理器执行的其它指令要么在这对指令之前执行，要么在这对指令之后执行，那么就可以认为这对指令的执行具有原子性，即具有原子性表象。

LL/SC指令对是一种非常有用的实现原子性表象的一种机制。它包括一种特殊的加载指令链接加载（load linked , LL）和条件存储（store conditional, SC）两种指令。对于链接加载的指定位置，如果其内容在对同一位置执行条件存储之前发生了改变（例如在缓存一致性协议中，高速缓存块的状态为I状态），或者在条件存储之前发生了其它可能违背原子性表象的事件（例如上下文切换）时，条件存储就会失败，指令对及之间的指令重新执行。由于条件结果失败，因此处理器的执行结果没有可见的影响。因此，这些可能违背原子性表象的事件就相当于在这对指令之前执行，由此保证了原子性表象。

由于不需要为原子指令支持原子性，因此LL/SC指令对的硬件实现的代价不高。硬件上常利用链接寄存器来实现LL/SC指令对。在执行链接加载（LL）指令时，处理器除了加载相应的数据块之外，还会将数据块的地址存入链接寄存器中。当在链接加载和条件存储指令之间发生了违背原子性表象的事件（例如上下文切换或者高速缓存块失效）时，链接寄存器会被清空。在执行条件存储指令时，只有存储涉及的地址和链接寄存器匹配时（链接寄存器被清空时不会匹配），条件存储才能成功，由此保证了原子性表象。

### 2.3.3 利用LL/SC指令对实现锁

利用LL/SC指令对可以实现很多其它的同步原语，代码2-1展示了如何利用LL/SC实现锁。其中lockvar的值表示锁是否已经被申请，值为0表示未被申请，值为1反之。在申请锁的过程中，不断链接加载测试lockvar的值是否为0，若为0则加1后进行条件存储。若在条件存储前，其它处理器申请锁成功，那么由于高速缓存中的数据失效，链接寄存器被清空，条件存储失败，返回到链接加载的位置继续链接加载并测试。否则条件存储成功，锁的申请成功。释放锁和通常的释放锁是相同的。



代码2-1 利用LL/SC指令对实现锁

# 3 同步软件原语：Fetch\_and\_φ

## 3.1 热点问题及其性能影响

### 3.1.1 热点问题与fetch\_and\_φ的关系

热点（hot spot）：当大量处理器访问同一个共享变量时，被称为热点访问。被访问的共享变量被称为热点。Fetch\_and\_φ原语是一种原子的read—modify—write操作。显然当多个线程同时对一个内存地址中的内容进行Fetch\_and\_φ操作时会造成热点问题。当顺序串行的处理所有的Fetch\_and\_φ操作时，不仅会导致内存争用严重降低性能，而且还会导致互连网络中的树饱和（Tree Saturation）[2]，从而阻塞热请求和常规请求。

### 3.1.2 树饱和

假设N是系统中处理器的数量，同时系统中内存模块的数量也是N，但是只有一个内存模块是热点内存。若一个周期每个处理器发出的访问请求个数是r（0<=r<=1），h表示在所有的访问请求中热点访问请求所占的比例。因此在每个周期内有Nrh个热点请求，与热点内存模块相关的正常请求有r(1-h)个。每个周期总共有Nrh+r(1-h)个访问请求与热点内存模块相关。当Nrh+r(1-h)的值等于连接处理器和hot memory的链路上最薄弱的链路的处理能力时，就会出现饱和现象。若每个处理器在每个周期只能处理一个访问请求，则每个处理器的最大吞吐量是H=1/(1+h(N-1))，总的有效内存带宽为B=N/(1+h(N-1))。根据此公式可以的得到处理器数量与吞吐量之间的关系，绘制成如下图：

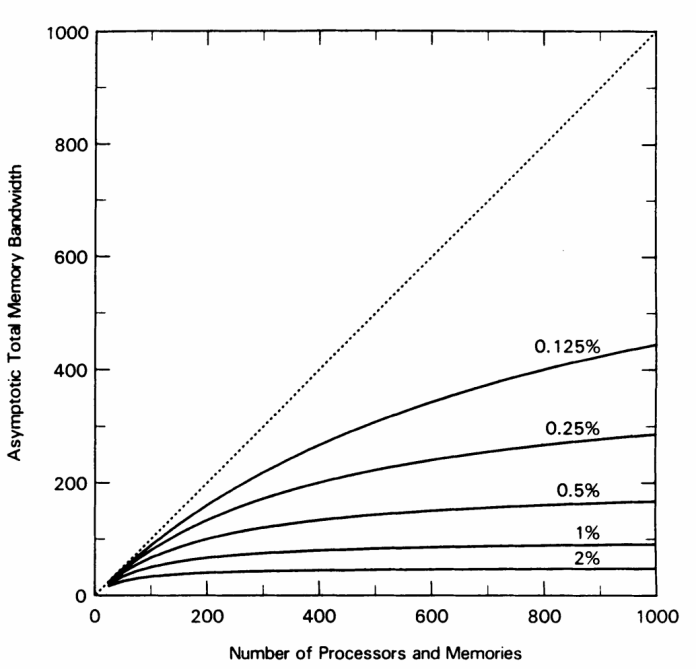


图3-1 处理器数量与总带宽关系图

发现即使热点流量只有1%，有1000个处理器和内存模块，总的内存带宽会被限制在10%以下。这种现象被称为饱和树是因为，最先出现饱和的节点是与hot memory相联的交换机的队列最先饱和，然后就是与饱和交换机相联的两个交换，依次类推，呈树状结构饱和。

## 3.2 硬件fetch\_and\_φ原语实现——消息组合技术

消息组合技术[2]是硬件实现Fetch\_and\_φ原语的技术，消息组合的工作原理是在经过每个交换节点时，检测指向相同内存位置的内存请求消息的发生情况。在交换节点上，这些消息被组合为单个消息。合并发生的事实记录在每个交换节点的等待缓冲区中。当对合并消息的应答到达合并该消息的节点时，将生成多个应答以满足多个单独的请求。由于在连续的交换阶段，组合的消息本身可以被组合。然而使用这种方式实现Fetch \_and\_φ原语需要对交换机进行改进，硬件成本花费巨大，因此通常不使用这种方式实现Fetch \_and\_φ原语。

## 3.3 硬件fetch\_and\_φ原语实现——软件组合树

### 3.3.1简单的软件组合树

软件组合树的核心思想就是将对一个热点的操作分散开来，变为对多个个热点的操作。之后在用树的结构将这些热点组织起来。下图将以一个例子说明如何用简单的软件组合树实现Fetch\_and\_Sub操作。

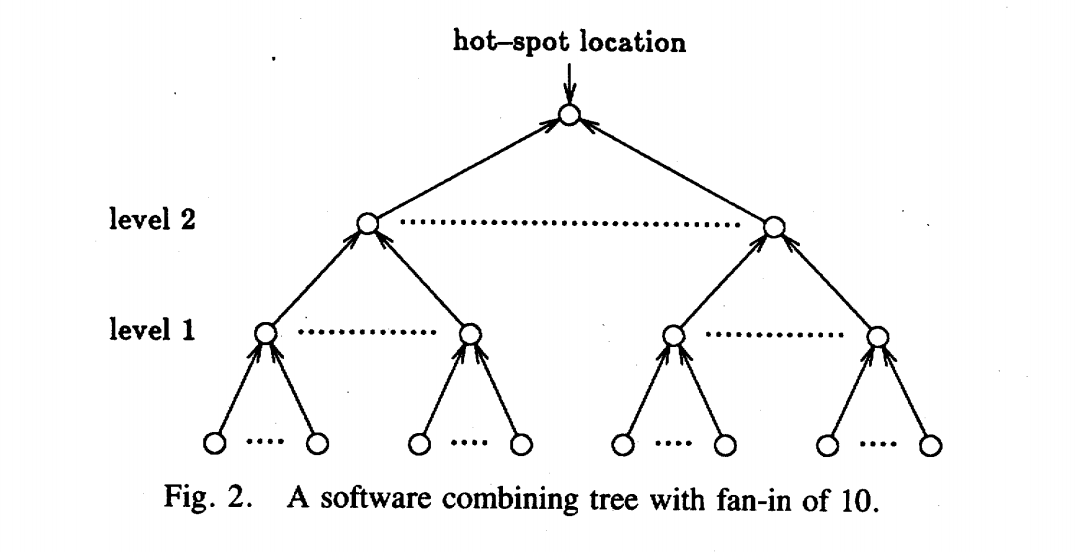


图3-2 软件组合树

假设现在有一个计数器N=1000，每个处理器将其减一，直到减为0.现在用软件组合树的方式来解决这个问题 。我们再建立一个单一的变量，而是建立一个变量树，将每个变量分配给不同的内存模块，如上图所示. 如果N=1000，假设树的入度为10，我们有111个热点变量，每个值为10。我们将处理器分成100组(每组10个)，每组共享树level1一个变量。当每个组中的最后一个处理器将其变量减为0时，它然后在父节点中减该值。因此，总访问次数从1000增加到1110，但是现有在111个热点，每个热点只有10次访问，而不是一个热点有1000次访问。因此，即使总的访问增加，吞吐量率和带宽依然得到显著改进。

### 3.3.2 二叉组合树

在3.3.1中描述的软件组合树方式中每个处理器只发出一个请求来减少一个计数器，当所有请求都完成时，计数器的值将为零。然而现实中的Fetch\_and\_φ操作要比这个例子复杂的多。例如1.处理器可能重复发出请求。2.任何时间内，每个处理器发出请求的数量是不可预测的。因此论文[1]中提出了一种更为复杂的二叉组合树。与3.3.1相比核心思想依然没变。不过为了处理更加复杂的情况，增加了很多字段来记录过程中的一些数据。

在这个算法实现的Fetch\_and\_Add中，一组进程可以在在任意时间提出增加共享计数器的请求。counter变量被构造成一个二叉组合树，每个节点上存储单独的增量用于组合，而实际值存储在根节点上。

每个节点包含五个字段：

1.Status：表示该节点的状态，有三种状态，分别为：

Free：表示此节点还未声明进行任何操作。

Combine：表示此节点要进行组合操作。

Result：表示此节点包含一个结果。

2.Wait\_flag：指示进程是否在该节点等待结果。

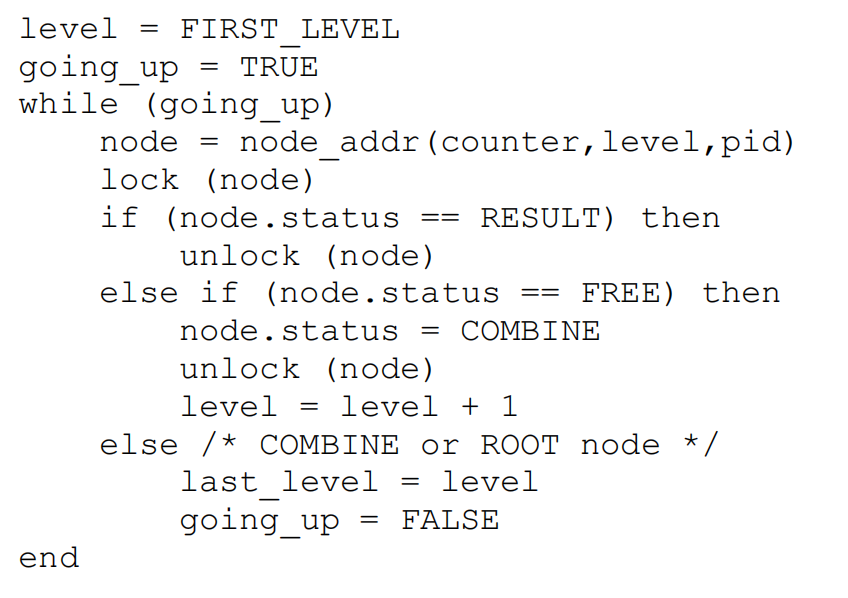
3.first\_incr：包含请求节点的进程的子树打算将计数器递增的数量。

4.Second\_incr：等待进程打算将计数器递增的数量。

5.Result：要分发给子树的计数器值。

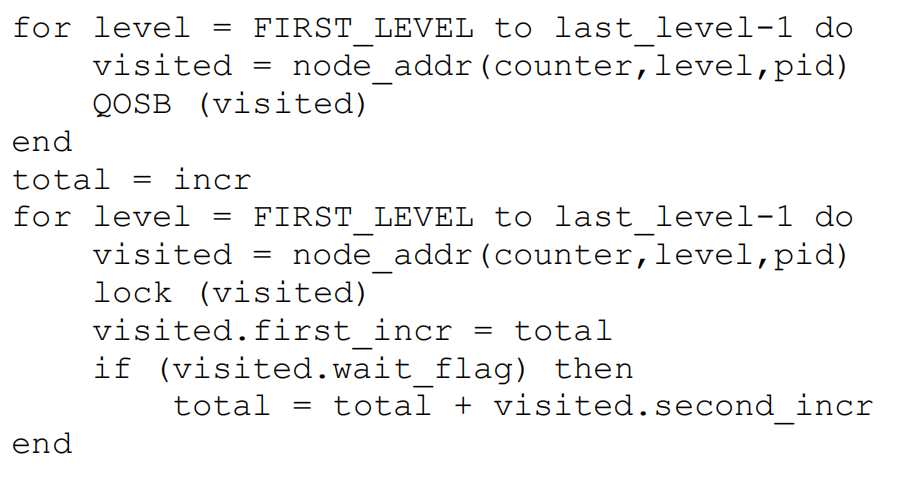
该算法分为四个部分：

1. 如果一个进程要进行Fetch\_andAdd操作，则自底向上更改free节点为COMBINE节，直到找到ROOT节点或COMBINE节点。 如果遇到result节点，则自旋等待其状态更改后再继续上树。



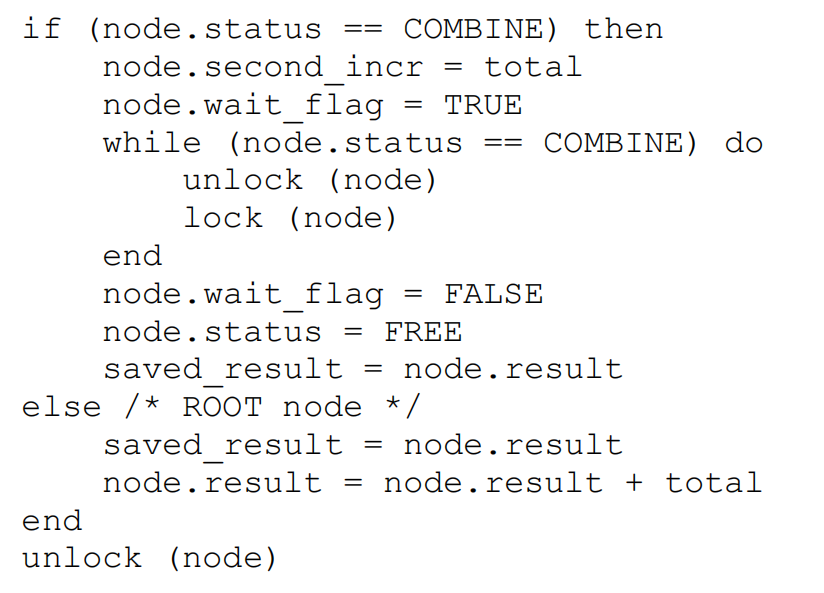
代码3-1 二叉组合树算法第一部分

1. 锁定之前访问的每一个节点，进行组合操作，更新first\_incr字段和total值，直到结果被分发，节点才会被解锁。



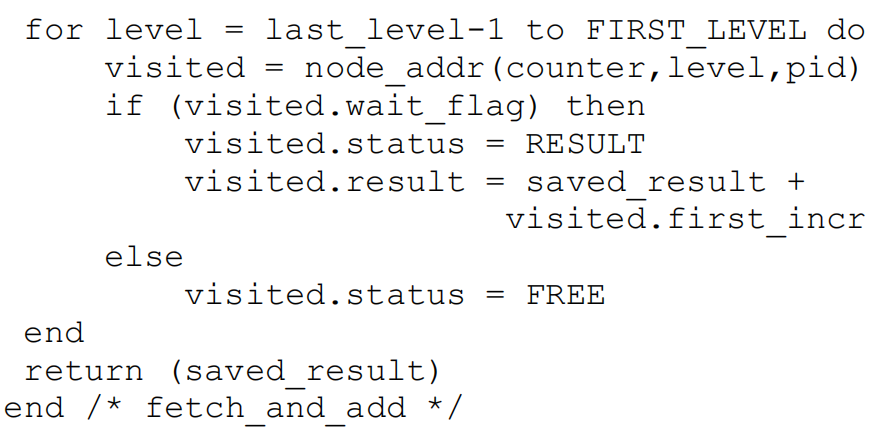
代码3-2 二叉组合树算法第二部分

1. 如果第一部分在COMBINE节点上停止，则将此进程的total放 入 节点的 second\_incr 中，设置wait\_flag，并等待节点状态更改为result；如果第一部分停止在ROOT节点，则保存结果并将total添加进去（fetch and add）。



代码3-3 二叉组合树算法第三部分

1. 释放节点或者分发结果



代码3-4 二叉组合树算法第四部分

如下图所示二叉组合树是进程4进行Fetch\_and\_Add操作示例：

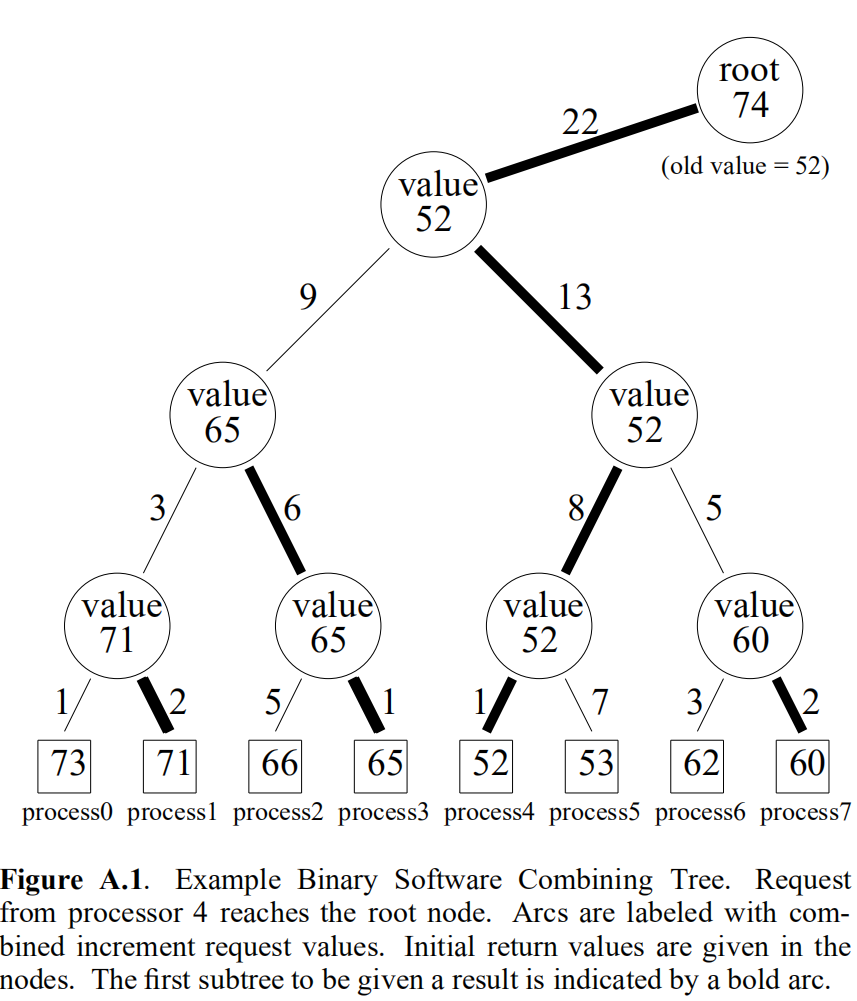


图3-3 二叉组合树举例

# 4 自旋锁Spin lock

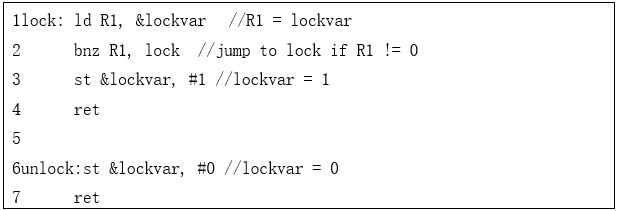
## 4.1 自旋锁简介及性能指标

前面讲述了如何在硬件层次上支持同步原语，本节将在软件层面上分析如何实现同步操作，针对四大类同步锁，逐个分析在多处理器下无竞争延迟、锁等待通信量、锁释放通信量、公平性、空间占用这些指标的情况，以比较不同锁的优劣，分析在不同场合下优先选择什样的锁。

本节将要介绍的锁一个有四大类，分别是TS、TTSL、Ticket和ABQL，前两种是较为低级的锁，后两者可以是基于前两种基础锁构建出的实现公平性的锁。而处于为了提升某方面执行效率和减少开支的考量，又在TTSL锁基础上提出了带指数退避机制的锁（TTSL exp.backoff），和在Ticket锁基础上提出α退避机制锁（Ticket α.backoff）。在介绍这四大类锁之前，先对设计指标进行简要介绍。

1. 无竞争延迟，指当线程间没有竞争时，获取一个锁所花费的时间。
2. 锁等待通信量，当锁被一个线程所持有的时候，其他线程为了获取这把锁所产生的总线上的通信量。
3. 锁释放通信量，线程释放锁时，为了通知其他线程锁已释放所产生的通信量。
4. 公平性，该锁是否保证公平性，即先竞争的线程一定比后竞争的线程先获得锁。
5. 空间占用，为了实现该锁，所造成的程序非必须的空间开销。

为了让设计的锁能够实现同步操作，有必要了解（回顾）一下硬件层面所提供的支持。首先来看一种不正确的锁的实现，代码如代码4-1所示。



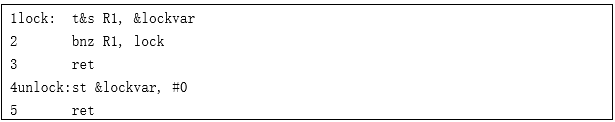
代码4-1 错误的同步方案实现

上述代码如果要正确执行，必须保证读取和修改lockvar值的操作是原子的，即要么全部执行，要么都不执行，在该场景下，即只能有一个线程执行成功上述语句。锁本身的原子操作需要依靠硬件提供的相应机制来实现。幸运的是，缓存一致性协议以及总线调度机制可以帮助实现这类功能。仍然以上述代码为例，若线程0加载了lockvar变量，为了原子的执行后续操作，排他的访问、修改该变量，首先将其余线程有关该变量的缓存块置无效，同时锁总线/预约总线，这样当其他线程想要访问该变量，只能再次从内存中读取变量，然而总线已上锁，因此就只能等待线程0操作完成。总的来说，对块的其他请求进行延迟直到操作完成以组织块被“偷走”，进而保证原子性。下面将详细介绍这四大类锁及其变体。

## 4.2 各类锁

### 4.2.1 TS锁

该锁基于test-and-set原子指令实现，是利用上述机制最简单、最基础的一种锁实现。Test-and-set指令原子的执行如下步骤：读取lockvar变量的值到寄存器R1中，且该读取是排他读指令（BusRdx），比较R1中的值是否为0，如果是0，则可以获取锁，并将lockvar置1，如果是1，则无法获取锁，并尝试重新获取锁。排他读和总线预约机制保证了上锁操作本身的原子性，进而保证锁的正确性。TS锁实现如代码4-2所示。



代码4-2 TS锁实现代码

接下来以图示的方式进一步阐述锁获取的过程，并分析其中的开销。首先是锁未被占用的情况，如图4-1所示。

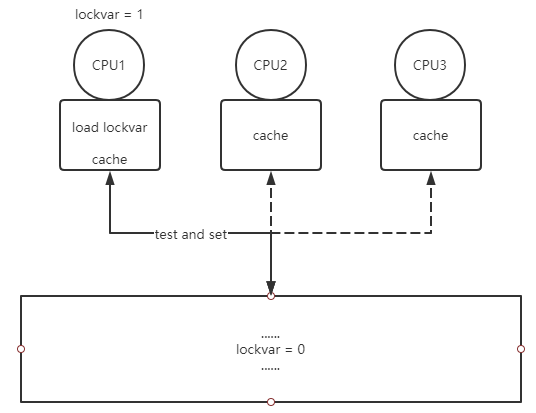


图4-1 TS锁初始竞争

此时，内存中lockvar变量值为0，cpu1中线程1首先尝试获取锁，那么加载了该变量，然后置其余cpu中有关该变量的缓存块为无效，并预约总线，锁获取成功后，取消总线独占。接着，图4-2所示是其余线程在锁占用是的竞争图示。

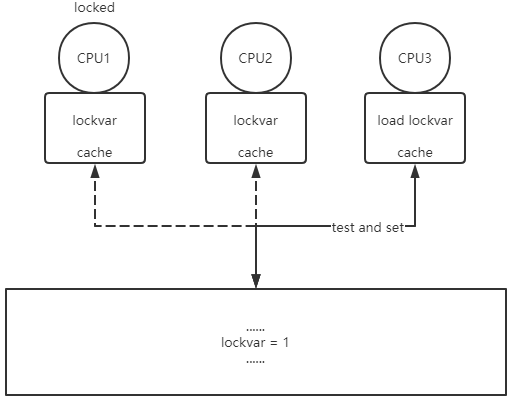


图4-2 TS锁等待期间竞争

例如，cpu3中线程此时想要获取锁，首先读取lockvar变量值，然后置其余cpu中有关该变量的缓存块为无效，并预约总线，此时检测到lockvar值为1，即无法获取锁，之后取消总线独占。完成一轮竞争。

通过以上分析得出，TS锁实际上是一种循环+排他读的一种低级锁，这种锁的执行效率地下，其中存在这很多不必要的操作，造成了通信的浪费。如图4-3所示是TS锁竞争的流程图，每个锁获取都会导致其他缓存块的失效，而不管这次获取锁的操作是否成功，加上不断循环操作，导致锁即使已被占用期间，也会导致大量的通信开销。

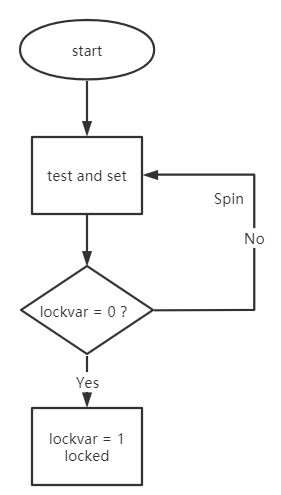


图4-3 TS锁竞争流程图

如图4-4所示是三个线程在竞争锁时各个时刻缓存块对应的状态以及说明。在P1成功获取锁之后，P2、P3甚至更多的线程都会不停的对lockvar变量执行test-and-set操作，因此后续的过程中，有很多的M位，代表失效缓存操作，如果有P个处理器竞争该锁的话，那么就会有O（p2）的通信量。同样，锁释放操作也会导致O（p）的通信开销。另外，该锁还有一个致命的缺点，即当未持有锁线程不停的执行test-and-set操作时，有可能导致总线饱和，特别对于已经获取到锁的线程，如果因为这种饱和导致业务操作受阻，以及释放锁对于lockvar变量的复0也会受阻，进一步加剧了锁竞争。

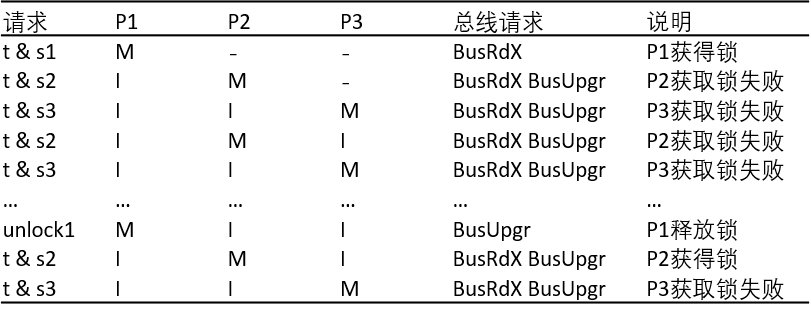
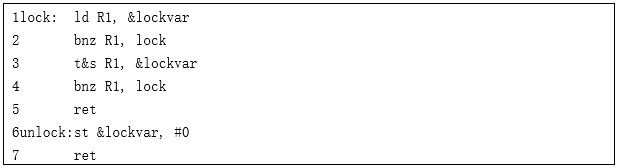


图4-4 TS锁竞争及总线状态

### 4.2.2 TTSL锁

通过以上分析，可以很容易看出TS锁的缺点，即等锁期间的通信量过高，且这部分操作是不必要的，通过分析得出，在等锁期间，lockvar变量恒为1，且除了释放锁操作外，没有线程可以修改该变量的值，那么缓存中的数据始终保持有效。因此等待锁期间，就没有必要对已经缓存的变量重新到内存中读取，只需要在执行test-and-set操作之前，先检查缓存中的lockvar变量是否发生改变（由持有锁线程释放锁时导致的缓存失效），不断循环检查缓存中lockvar变量值。当锁释放后，缓存失效，重新读取变量，如果变为0，则继续执行test-and-set操作。TTSL锁竞争代码如4-3所示。



代码4-3 TTSL锁实现代码

与TS锁代码的区别较为明显，等锁期间，不断读取lockvar所在地址，而不是尝试修改lockvar，这样就阻止了无用的失效和重新读取操作，从而使得等锁期间的通信量大大降低，不过由于多了一次检查操作，使得无竞争期间获取锁操作的开销略微增大。

如图4-5所示是锁竞争期间各线程的状态示意，于TS相比，cpu2、cpu3不再去从内存中获取lockvar变量，而是从本地缓存中检查lockvar变量。从表中也可以看到，原来M状态被大面积修改为S，减少了排他读操作，使得总线通信量大幅减小。

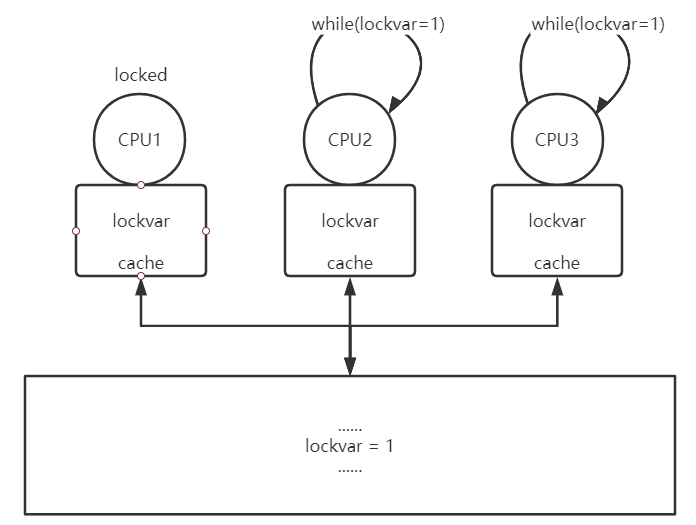


图4-5 TTSL锁等待期间竞争

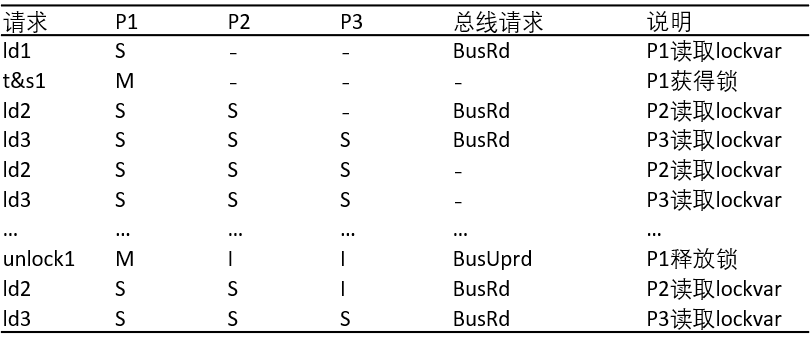


图4-6 TTSL锁竞争及总线状态

然而TTSL锁在释放的时候同样也会产生O（P）的通信量，这种现象称为总线风暴，竞争锁的线程数量越多，总线风暴现象越严重。改良的版本是带指数退避机制的TTSL锁（TTSL exp.backoff）。

该方案的有效性是显而易见的，如果要避免总线风暴，就需要在锁释放的一瞬间，仅有少数线程去竞争该锁，如图所示是带指数退避机制的TTSL锁的锁竞争流程图，当循环自检缓存变量lockvar为1时，退避1个时间单位进行下次检查，如果再次为1，退避2个时间单位，依次4、8、16……退避在于当抢不到锁时，进行一小段时间的休眠，将cpu时间片让给其余线程，指数机制在于如果接二连三的抢不到锁，则认为下次大概率仍无法获得锁，则将休眠时间加长。

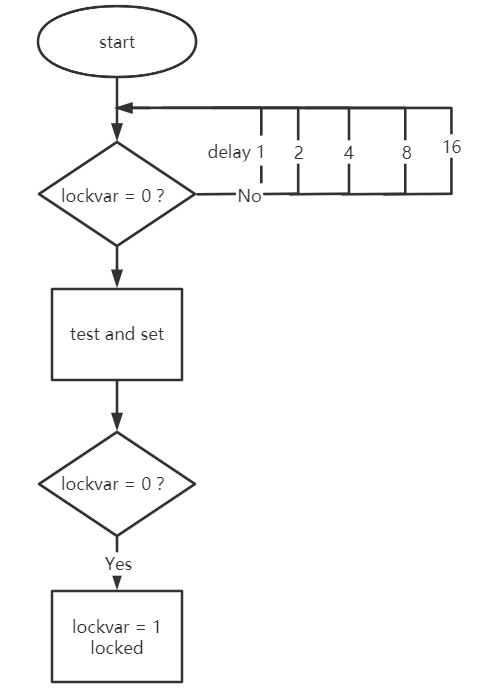
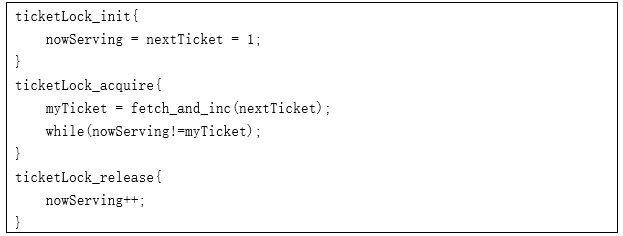


图4-7 指数退避的TTSL锁竞争流程图

指数退避机制很好的规避了总线风暴现象，但仍存在一些细节问题。退避的机制怎样实现才能确保不产生浪费？考虑一个场景，如果后期退避的时间单位已经很大，恰巧刚一轮询完，锁就被释放了，那么此时再进行长时间的等待就变得不合适了，因此，在达到最大等待时间后，可以考虑在较小的等待时间范围内重新随机选取一个。另外，还有公平性的问题，试想如果一个线程每次在较短的等待时间后就能获取到锁，那么那些没有获取到锁的线程的等待时间依然会变得很长。会陷入到越拿不到锁就越不容易拿到锁的困境。

### 4.2.3 Ticket锁

TTSL解决了TS锁等锁期间通信量的问题，但暴露出公平性的问题。公平就是要保证参与竞争的锁都能在有限时间内成功获取到锁。通过对前面两种锁的分析，可以看出它们都无法保证公平性，而Ticket锁提出了一种票证机制，用于保证公平性，使得线程获取锁的顺序与尝试去竞争锁的顺序一致。为了实现公平性，以一种队列的形式为每个竞争锁的线程去分配获取位，这样获取锁的线程就会依次被安排在队列的后面，前面的线程释放了锁后，其才能成功获取锁。如下是代码部分。



代码4-4 Ticket锁实现代码

为了实现这种机制，引入了nowServing和nextTicket变量，前者用于指明当前正在执行的线程排队号，后者用于给线程分配排队号，通过fetch\_and\_inc互斥的对nextTicket变量进行自增操作，其底层实现可通过TS/TTSL锁来实现。如图所示是锁竞争期间，不同cpu线程的执行情况，此时cpu2、cpu3自旋检查nowServing是否等于myTicket，如果不相等，则认为无法获取到锁，将不停检查缓存中的myTicket变量，这一步类似于TTSL。当cpu1上的线程释放锁后，对nowServing进行+1，其余cpu遭遇一次缓存失效，更新nowServing变量后，cpu2的线程发现可以获取到锁，则尝试获取锁，cpu3及以后的线程等待。

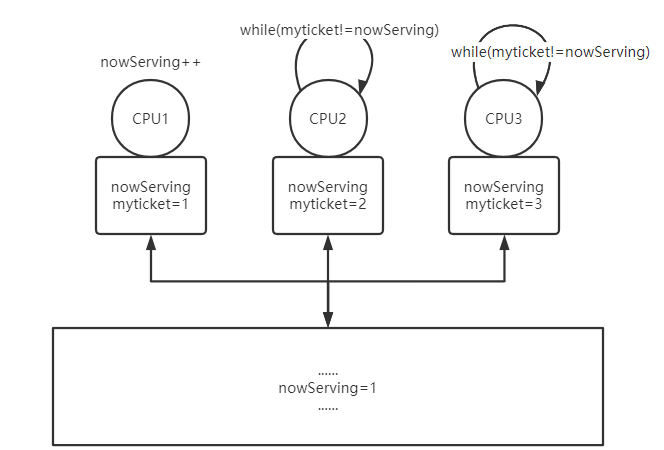


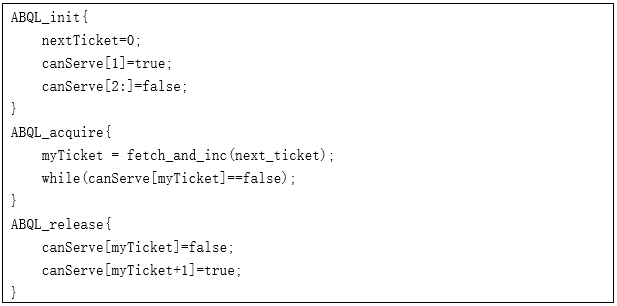
图4-8 Ticket锁等待期间竞争

然而，Ticket锁检查nowServing的这一步操作仍然存在总线风暴现象，其余线程在持有锁线程释放锁的一瞬间，都会去更新nowServing值，同样，等待的线程数量越多，总线风暴情况越严重。

为了解决该问题，参考TTSL退避机制，Ticket锁也可以引入一种退避机制，只不过退避方式有所不同，考虑到Ticket锁是一种强调公平性的锁，那么先竞争的锁需比后竞争的锁更快的抢占到锁，因此退避时间应当与（myTicket - nowServing）有关，将这种锁称为基于α.backoff的Ticket锁，α为退避因子，和临界区执行时长有关，这样的设计，既保证了公平性，也不会出现锁可用的情况下线程仍然在等待。需要注意的是，如果退避时间结束后，发现仍无法获得锁，则重新计算（myTicket - nowServing）的值，生成新的退避时间。

### 4.2.4 ABQL锁

这是避免总线风暴又能保持公平性的锁，与Ticket锁实现机制不同，并没有采用退避策略，而是改变自旋对象，等锁期间，其余线程并不是检查同一个变量，而是检查各自的变量是否变为1，该机制实现的代码如4-5所示。初始化时，nextTicket复0，用于给其他线程互斥的分配排队号，canServe初始时刻只有第1个元素为true，代表第一个线程可以立即获得锁，其余为false。而在释放锁的时候，释放锁线程将自己的排队号对应位置置为false，并将下一个排队号对应位置值为true，代表可运行。



代码4-5 ABQL锁实现代码

如图4-9所示是ABQL锁下cpu线程等锁期间的运行状态，此时每个线程都只会检查各自位置的变量，也就不会产生总线风暴。锁释放所产生的通信量大小也仅为O(1)。但需要注意的是数组的实现形式需要保证各排队号对应的位置刚好位于不同的缓存块上，否则修改canServe[i]对应的值将有可能造成若干个cpu缓存块的失效，通常的做法是采用链式结构。

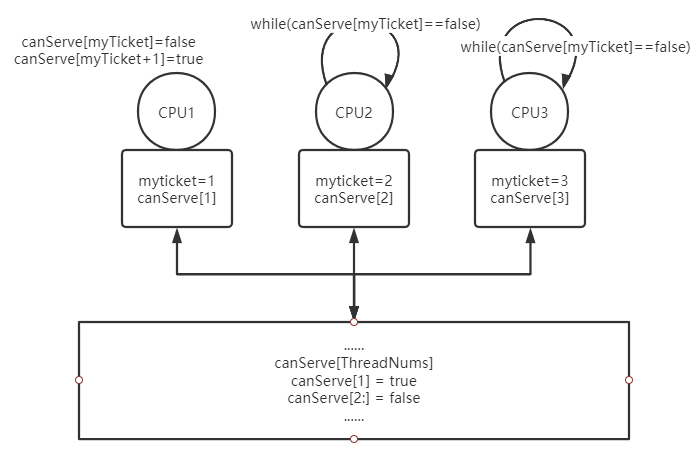


图4-9 ABQL锁等待期间竞争

## 4.3 总结

最后，总结一下本节所讨论的四大类锁，TS锁实现最简单，无竞争获取锁的延迟最小，但等锁期间的通信量巨大，且不必要。TTSL锁相较TS锁，增设缓存块本地自检，无竞争获取锁延迟稍微增加，但等锁期间通信量极大削减。它的改进版本带指数退避机制的TTSL锁，消除了总线风暴，但退避机制设置不当极易造成锁竞争的不公平。Ticket锁通过为每一个线程分配一个排队号保证公平性，但仍会造成总线风暴问题，改进版本α退避机制的Ticket锁消除了总线风暴问题，退避时间与线程排队号和当前服务号的差值有关。ABQL锁从另一种途径避免了总线风暴问题，方法是改变自旋对象。如图所示是上述所有锁的指标对比。

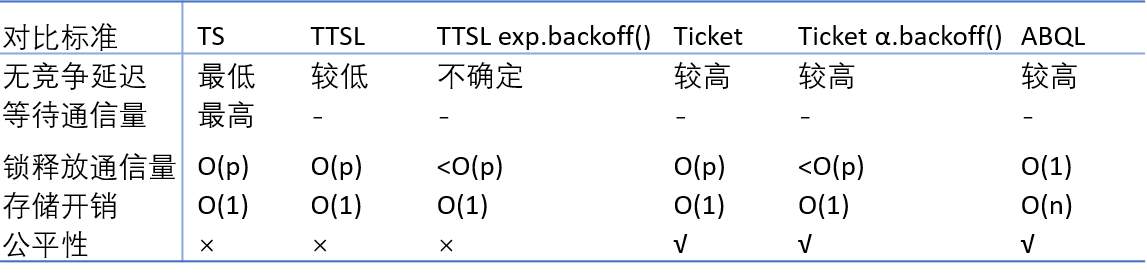


图4-10 各类锁及其变体总结对比

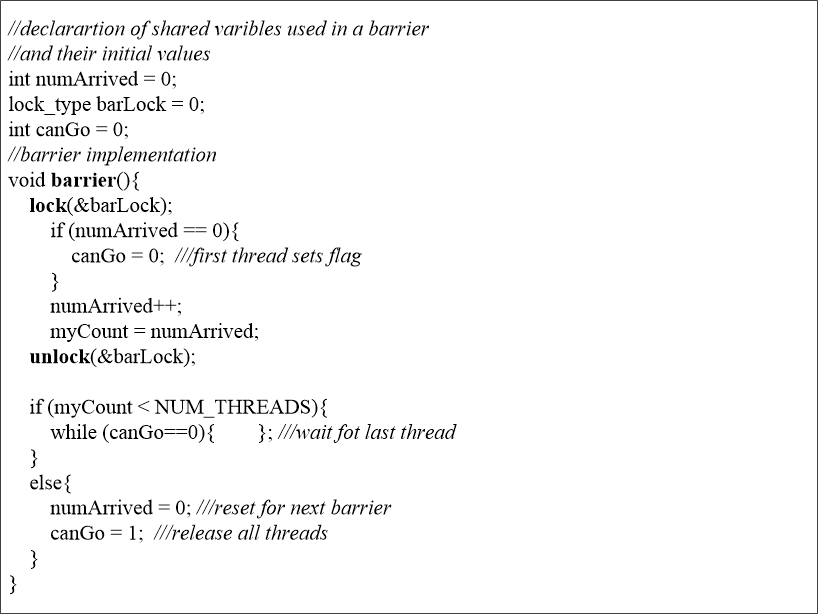
# 5 栅障Barrier

## 5.1 栅障的目的

栅障定义了一个时刻，在此时刻所有处理器在运行中都处在已知的状态点上。不同于锁的概念，栅障并不强调独占性，而是进入下一阶段前的最后一步。很多资料中将同步看作一个瞬间完成的原子动作，但不去解释当同步从两个进程的操作扩展至10个、100个进程时的同步过程。下面将详细介绍几种栅障的设计思想。

## 5.2 软件栅障

### 5.2.1 集中式栅障

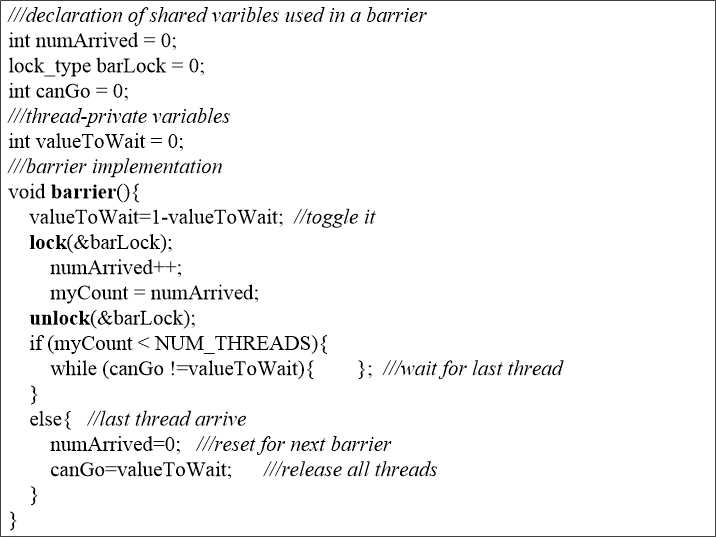
集中式栅障是最简单也是最容易想到的同步方式。每个处理器通过更新少量共享状态以指示其到达，然后轮询该状态以确定所有处理器何时到达。一旦所有的处理器到达，每个处理器都被允许继续通过屏障。代码5-1是这个算法的伪代码。

代码5-1 集中式栅障算法伪代码

然而，如果存在两个连续的栅障，上述算法在某些情况下会出现问题。例如，当通信延迟较高时，进程不知道自身得到的值是否是当前栅障的状态，还有可能是上一个线程的状态，在极端的情况下，会导致死锁的情况发生。因此我们需要对上述算法进行改进。

第一种改进方法是在进程进入新的栅障并重新初始化之前，要先等待其它所有的进程从前一个栅障中释放。在实现的过程中，这一改进等同于每次栅障需要两次同步后才可以继续进行。这一改进方法需要将标志数量翻倍、通信的过程也将翻倍。当进程数量增多时，开销会指数增长。

第二种改进方法是在相邻的两个栅障中使用不同的值代表释放。到达的处理器递减计数，然后等待，直到sense的值与前一个barrier中的值不同。最后到达的处理器重置计数并颠倒感知。连续的屏障不会相互干扰，因为所有计数操作都在sense被切换以释放等待的处理器之前发生。



代码5-2 集中式翻转栅障算法伪代码

第三种改进是使用两个共享计数器和一个处理器私有的双状态标志。私有标志选择要使用的计数器;连续障碍使用交替计数器。

集中式栅障的缺点是共享的位置只有一个。因为处理器实际上不会同时到达栅障，所以忙碌等待访问的数量通常会远远高于最小值。然而，在没有一致缓存的机器上，对共享位置的忙碌等待访问可能会产生不可接受的内存水平和互连网络的争用。对于基于广播的多处理器，这些访问则不是问题。共享标志或感知变量被复制到每个等待处理器的缓存中，因此可以在没有任何网络流量的情况下满足后续的忙碌等待访问。此共享变量仅在达到屏障时才写入，从而导致所有缓存副本的一次广播无效。然后所有忙碌等待的处理器获得该变量的新值，并能够继续处理。

为了减少因忙于等待屏障标志而造成的互连网络流量，可以采用自适应后退方案，使处理器在连续轮询操作之间延迟几何增长的时间。在许多情况下，这种指数回退可以显著减少实现栅障所需的网络通信量。但是，随着网络流量的减少，栅障的延迟也会增加。处于长延迟中的处理器不会立即注意到所有其他处理器已经到达。因此，它们离开障碍物的时间被推迟，这反过来又推迟了它们到达后续障碍物的时间。

同时，对于超过256个处理器的系统，在单一标志上运行的屏障，后退策略的效用有限。在这种大规模系统中，由于互连网络中的冲突导致处理器重复访问，每个处理器的网络访问数量急剧增加。这些观察结果表明，即使使用自适应后退策略，集中式阻挡算法也不能很好地适应大量处理器。

### 5.2.2 组合树栅障

为了减少同步变量的热点争用，就像多级互联网络中的硬件组合一样，软件组合树的作用是将对同一共享变量的多个引用收集为单个引用，其效果与单个引用的组合效果相同。期望作为多个并发访问目标的共享变量被表示为“变量树”，树中的每个节点被分配给不同的内存模块。处理器被分成几组，每一组分配给树的每个叶。每个处理器更新其叶节点中的状态。如果它发现自己是组中最后一个执行此操作的处理器，它将继续沿着树向上移动，并更新父处理器，以将集体更新反映给子处理器。按照这种方式，后来者的处理器最终将更新传播到树的根节点。

对于最基本的组合树栅障来说，也可以使用翻转来避免连续的栅障的重叠，使得单次栅障只需要修改一次共享存储信息，并且用简单的读取替换了组合树的原子指令，因为没有返回真正的信息。每个处理器从组合树的一个叶开始，并递减其叶的计数变量。到达树中的每个节点的最后一个处理器继续向上一级。到达树的根的处理器开始对标志进行反向更新。一旦它被唤醒，每个处理器就会通过树回溯它的路径，解除路径上每个节点上它的兄弟节点的阻塞。通过实验表明，这样的方法可以显著缓解集中式栅障的拥塞现象。

组合树屏障的主要缺点是，它要求处理器在不能在确定的内存位置上读取，而其他处理器也在这些位置上读取。一般情况下，处理器将在远程位置上进行读取，从而导致不必要的争用互连网络带宽。而在基于广播的缓存一致性机器上，处理器可能会获得它们在其上的树节点的本地副本。

### 5.2.3 蝴蝶栅障

蝴蝶栅障是一种对称的算法，每个进程每一步执行相同的操作。在经过次同步后，所有进程均知晓其他进程的同步状态，并可以继续向下执行。

在第k轮的同步过程中（k从0开始计数），进程i与进程进行同步，每次同步都需要获取对方的状态信息并将自己的状态信息提供给对方，即需要远程更改一个标志、本地更改一个标志。每次同步结束后，进程可以知晓个处理器的到达状态。在经过次同步后，所有的处理器进程均知道所有的进程的到达状态，从而可以继续进行。

在每次同步时，需要两个进程分别读取一个远程状态，因此需要2次通信过程。在每个进程内部需要一个数组来存储所有处理器的状态，因此共需要的空间需求。

### 5.2.4 散播栅障

散播栅障是蝴蝶栅障的一种改进。在第k轮的同步过程中（k从0开始计数），进程i与进程进行同步，每次同步仅需将自己的状态更改，并读取对方的状态，因此每次同步只需要一次通信。同样经过次同步后，所有的处理器进程均知道所有的进程的到达状态，从而可以继续进行。

散播栅障的好处在于不需要远程地更改标志，这大大地减少了同步中的通信开销。同时也可以通过翻转来进行优化。

### 5.2.5 比赛栅障

比赛栅障是对组合树栅障的一种改进。在组合树栅障中，由于到达顺序的不同导致每次继续向上传递的进程是不固定的，这种不确定性会额外增加通信和存储开销。比赛栅障将进程进行静态分组，第k轮(从0开始计数)中，i设置一个标记，等待进程j，其中i恒等于 ，。处理器i然后退出竞赛，并忙碌地等待一个全局标志，以通知屏障已经实现。处理器j参加下一轮比赛。一场完整的比赛包括回合。处理器0在比赛结束时设置一个全局标志。因为所有处理器都在忙着等待一个全局标志，因此这一算法适合使用广播来保持缓存一致性的多处理器。然而，在缺乏一致缓存或限制高速缓存线复制程度的机器上，它们将导致严重的互连流量。

## 5.3 理论对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 集中式栅障 | 组合树栅障 | 传播栅障 | 比赛栅障 |
| 关键路径长度 |  |  |  |  |
| 网络事务的总数(无广播) |  |  |  |  |
| 网络事务的总数(有广播) |  |  |  |  |
| 空间需求 |  |  |  |  |
| 给定原子操作的可实现性 | 原子递增或递减指令 | 原子递增或递减指令 | 普通读写的原子性 | 普通读写的原子性 |

## 5.4 硬件栅障

上述的所有算法均是基于软件层面的实现，可以应用于任何并行的程序中。但是经过对比可以发现，不论如何改进算法，算法需要的时间和空间都将随着进程的个数逐步增加。 而硬件栅障则可以完美地解决这一问题。但是由于硬件栅障需要对不同的处理器进行单独布线，所以需要的成本很高，更适用于大型系统或超级系统。

硬件栅障通过电平的高低来象征该处理器是否到达栅障。最终通过“与”操作汇总到一个电线上，所有处理器都对信号进行监控。当信号值为高电平时，意味着所有处理器均到达，处理器可以立刻接受到该信号从而继续进行下面的步骤。这使得延迟可以达到最低。为了降低导线的传输距离以及避免物理层的信号干扰，也可以使用树形结构来进行汇总，如图5-1所示。在IBM Bluegene超级计算机中就使用了这一方式，该机器中共有64000个节点，而一次的通信时长仅需要1.5μs。

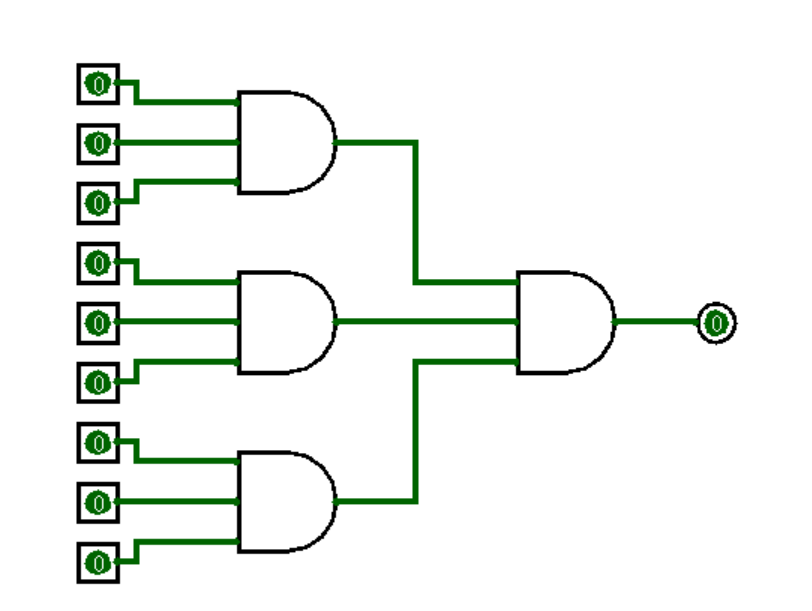


图5-1 硬件栅障实现示意图

# 结束语

本报告从同步机制所引发的问题，展开如何在多核共享存储系统中实现进程之间的同步。并分别从硬件和软件角度讨论同步原语的实现，为后文实现较高层次的同步锁、栅障做铺垫。从应用的角度上，分别讨论了锁和栅障的实现机制，从多个指标讨论不同实现版本的优劣，循序渐进，逐步完善。

最后，想说明的是，通过这次的论文阅读和分享报告，我们小组成员对同步这一章的知识体系又有了更深刻的认识，我们知道，这只是冰山一角，我们将对并行多核体系结构下的各种问题持续钻研下去。另外，感谢栾老师给了我们这次展示的机会，学习了知识，锻炼了技能，也感谢同课堂的同学，给我们提建议，互相交流，探索更加深刻的问题。

# 参考文献

[1]J.R. Goodman, M.K. Vernon, and P.J. WoeSt. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In Proceedings of the 3rd International Conference on Architectural Support for Programming Languates and Operating Systems, pages 64–75, April 1989.

1. G. F. Pfister and V. A. Norton, " 'Hot-spot' contention and combining in multistage interconnection networks," IEEE Trans. Comput., vol. C-34, pp. 943-948, Oct. 1985.
2. Yew, P. C., N. F. Tzeng, and D. H. Lawrie, “Distributing Hot-Spot Addressing in Large-Scale Multiprocessors,” IEEE Transactions on Computers, April 1987, pp 388-395.
3. John M. Mellor-Crummey and Michael L. Scott Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors ACM Transactions on Computer Systems, Vol 9, No. 1, Pages 21-65, February, 1991.