第十二章　多核与多处理器

在第八章中，我们介绍了如何在多核硬件中使用合适的同步原语来保证程序的正确性。为了简化问题，多核硬件被简单地抽象为数个单核处理器。然而在实际场景中，多核处理器并非数个单核的简单叠加。多核硬件暴露给软件的除了多个可用的计算单元之外，还有一系列硬件特性。这些硬件特性将对软件的性能造成巨大的影响。此外，在实际使用场景中，除了单一处理器中可以拥有多个核心，有的系统还拥有多个处理器，因此展现出更加复杂的硬件特性。

本章将首先介绍在操作系统视角下，多核多处理器硬件的部分特性，包括：缓存一致性、内存一致性模型以及非一致性内存访问。然后，我们将讨论操作系统开发者应该如何利用这些硬件特性，以获得最大的性能提升。

12.1　缓存一致性

本节主要知识点

- ❑ 多核多处理器中高速缓存是什么样的？
- ❑ 如何保证这些高速缓存中数据的一致性？
- ❑ 缓存一致性对于操作系统开发者来说意味着什么？

12.1.1　多核高速缓存架构

当前主流的多核处理器均采用了共享内存：不同的核心共享相同的内存资源，核心间可以通过访问同一个地址来共享数据。然而由于访问内存耗时较长，共享内存不是直接将多核处理器连接到同一个物理内存，而是添加了**多级高速缓存**（Multilevel Cache）来缓存高频访问的数据。访问该缓存的速度远高

于访问内存的速度，因此使用该缓存可以降低访问内存的概率从而减少访存开销。

我们首先回顾一下单核处理器中高速缓存的架构。高速缓存中使用**缓存行（Cacheline）**作为最小的操作粒度，其大小往往为 64 字节。在多级高速缓存中需要读一个地址的值时，处理器将逐级查找高速缓存中是否保存了该地址对应缓存行。如果在任意一级高速缓存找到，处理器将直接读取该值，从而避免耗时的内存访问操作。而当需要写一个地址的值时，处理器有多种策略可选。**直写策略**（Write Through）会立刻将修改的值刷回内存（该值会同时保留在高速缓存中），而**写回策略**（Write Back）则会将修改的值暂时存在高速缓存中，避免高时延的内存写操作。只有在出现**高速缓存逐出**（Cache Eviction），或是 CPU 核心调用写回指令时，修改才会被更新至物理内存。

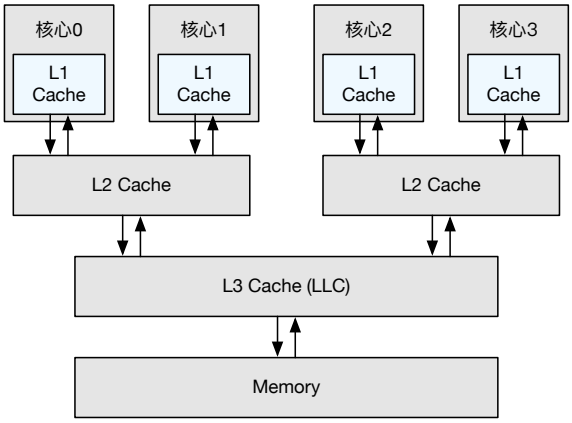

图 12.1: 多核环境下的高速缓存结构示例

而在多核处理器中，一个典型的多级高速缓存架构如图12.1所示。每个处理器核心均有自己的私有**一级缓存**（Level 1 Cache，L1 Cache）[1]，多个核心之间共享一个**二级缓存**（Level 2 Cache，L2 Cache），而所有核心共享**最末级缓存**（Last Level Cache，LLC）。这种设计能够保证每个核心访问私有高速缓存时达到更好的性能。在这种架构中，不同核心访问时延会依据缓存行所在位置有所差别。如图 12.1中的核心 0 访问本地二级缓存中的缓存行会远快于访问核心 2 与核心 3 共享二级缓存中的缓存行。我们称这种缓存架构为**非一致缓存访问**（Non-Uniform Cache Access，NUCA）架构。私有的高速缓存设计除了会导致非一致缓存访问以外，还会带来数据一致性问题。由于不同核心均拥有私有的高速缓存（如一级缓存），某一地址上的数据可能同时存在于多个

[1]通常 L1 Cache 进一步划分为单独的数据缓存（Data Cache）与指令缓存（Instruction Cache）。

核心的一级缓存中。当这些核心同时使用写回策略修改该地址的数据时，会导致不同核心上一级缓存中该地址数据不一致，违反了共享内存的抽象。为了保证私有缓存之间也能就某一地址的值达成共识，多核硬件提供了**缓存一致性协议**（Cache Coherence Protocol）。

12.1.2 目录式缓存一致性

缓存一致性协议有多种实现方案，包括**目录式缓存一致性**（Directory-based Cache Coherence）与**嗅探式缓存一致性**（Snoop-based Cache Coherence）。缓存一致性是由硬件保证的，其对于上层系统软件是透明的。这里通过介绍其中一种：目录式缓存一致性，来展现缓存一致性的基本硬件原理。

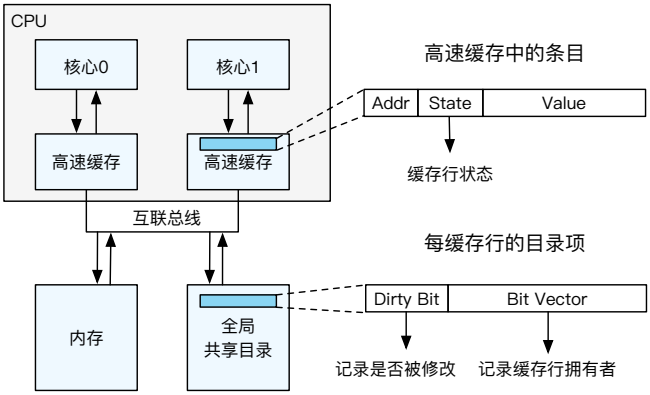

图 12.2: 目录式缓存一致性协议的简化硬件结构示意图

图12.2展示了使用目录式缓存一致性协议的简化硬件结构示意图。如图所示，每核心拥有自己私有的高速缓存，且所有核心可以通过互联总线访问共享内存。我们将通过讨论这种简化模型为读者展示缓存一致性的基本原理[2]。在多级高速缓存中，由于缓存之间关系更为复杂，因此需要遵守更多规则，但其背后的原理是一致的。私有高速缓存中的每条缓存行，除了其地址与值之外，还有单独的区域存储该缓存行的状态。这里我们主要介绍 **MSI 协议**，即缓存行的状态可以是**独占修改**（Modified）、**共享**（Shared）以及**失效**（Invalid）中的一个。这些状态的含义以及在不同状态之间迁移的条件将在下面详细介绍。除此之外，全局还有一个共享的目录，用于记录所有缓存行所处位置及其状态。每条缓存行都对应了该目录中的一个目录项，其包含两项内容：用于记录

[2]在简化模型中，单个访存操作导致的缓存一致性操作是原子的。实际硬件需要其他机制保证并发情况下缓存一致性的正确性。此外，该简化模型也不考虑缓存写回操作。

是否有处理器已经修改过这个缓存行的 Dirty Bit （1 表示被修改，0 表示未修改），以及用于记录缓存行的拥有者的 Bit Vector（拥有者对应位被置为 1，其余置为 0）。全局目录、高速缓存与内存三者均通过高速内部互联总线相连。目录式缓存一致性协议就是通过在全局共享目录中记录每个缓存行在不同核上的状态，确保对于缓存行的修改互斥且能够及时对所有核可见，从而保证了缓存的一致性。下面我们通过分析缓存行进行读写操作时在 MSI 三种状态之间的迁移，来详细介绍目录式缓存一致性协议的具体原理。

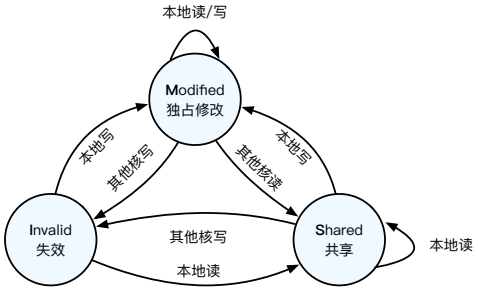

图 12.3: MSI 状态迁移

图12.3展示了 MSI 三种不同的状态以及当发生读写操作后状态的迁移。当一个核心需要访问的缓存行在私有高速缓存中时，该缓存行可能处于以下三种状态：

1. **独占修改**。这个状态代表着当前缓存行在全局只有本地高速缓存中这一份拷贝。因此当前的核心独占该缓存行，可以直接进行**读/写**操作，不会触发缓存行的状态变化。

2. **共享**。这个状态代表当前缓存行在全局可能存在多份拷贝，且本地的拷贝是有效的。因此当前核心可以直接**读**该缓存行。如果需要**写**该缓存行，则当前核心需要查找全局共享目录，找到所有拥有该缓存行拷贝的核心，并通知这些核心将缓存行状态转换为**失效**。之后再设置目录中该项的 Dirty Bit 为 1，更新拥有者的 Bit Vector。最后，将本地的缓存行状态转换为**独占修改**，方能对缓存行进行写操作。

3. **失效**。这个状态代表当前缓存行本地的拷贝失效，当前核心**不能直接读/写**该缓存行。如果需要**读**缓存行，则应当在目录找到拥有该缓存行的核心，向其索要该缓存行的数据，同时通知该核心将该缓存行状态改为**共享**。之后更新目录中的 Dirty Bit 为 0，并更新拥有者的 Bit Vector。最后，在本

核心中将拿到的缓存行设置为**共享**之后，方能读取该缓存行。如果需要**写**该缓存行，则需要通过目录找到所有拥有缓存行的核心，通知它们将该缓存行状态都改为**失效**。之后才能拿到该缓存行的数据，并更新目录中的状态。最后，将本地的缓存行状态设置为**独占修改**后，方能写该缓存行。

上面三种情况均为需要访问的缓存行已经在私有高速缓存中。而若需要访问的缓存行不在私有高速缓存中时，该核心将查看全局共享目录，检查该缓存行是否在其他核心上。如果其他核心拥有该缓存的拷贝，则处理流程同缓存行为**失效**一致。如果其他核心上也没有，则该次访问为**缓存不命中**（Cache Miss）。此时需要从内存中获取该缓存行，放到本地的高速缓存，同时更新全局的共享目录。而该缓存行状态将根据操作类型为读或写，设置该缓存行状态为**共享**或**独占修改**。现代硬件为了更好的性能，会采用更加复杂的协议，拥有更多的状态，同时也需要保证并发正确性并支持缓存写回操作，但其基本思路与我们介绍的简化模型一致。

下面通过一个具体例子，展示目录式缓存一致性的具体工作流程。图12.4中有三个核心，每个核心都有自己私有的高速缓存，分别标示为核心0、1、2高速缓存。这些核心共享一个全局目录项。图中的例子只关注地址 X 所在的缓存行，表项包含地址 X 中保存的值以及该缓存行状态。图中共展示了 $T_1 \sim T_6$ 时刻每个核心的高速缓存中 X 所在缓存行内容与状态，以及对应时刻全局共享目录项中关于该缓存行的元数据。T_1 时刻为初始状态，而 $T_2 \sim T_6$ 时刻，不同核心将执行该核心上方标注的指令。如在 T_2 时刻，核心 0 执行指令`ST X, 233`，该指令代表该向地址 X 中写入值 233，硬件将依照圆圈序号依次完成操作，保证缓存一致性。

小思考

在图12.4的例子中 T_3 时刻，核心 1 需要写地址X所在的缓存行，更新其值为 888 以覆盖旧的值，为何还需核心 0 将值传给自己？能否省略该步骤？

当多个核心对地址 X 执行写操作时，也不能由于会发生数据覆盖而省略传输旧缓存行的步骤。这是由于缓存一致性是以缓存行为粒度进行的，而一个缓存行的大小通常为 64 字节，写操作往往只作用于其中的一小部分。新的修改不会覆盖整个缓存行，依旧需要缓存行其他部分的旧值。因此需要等待核心 0 将整个缓存行发送给自己后（图中 T_3 时刻操作 3），才能修改。

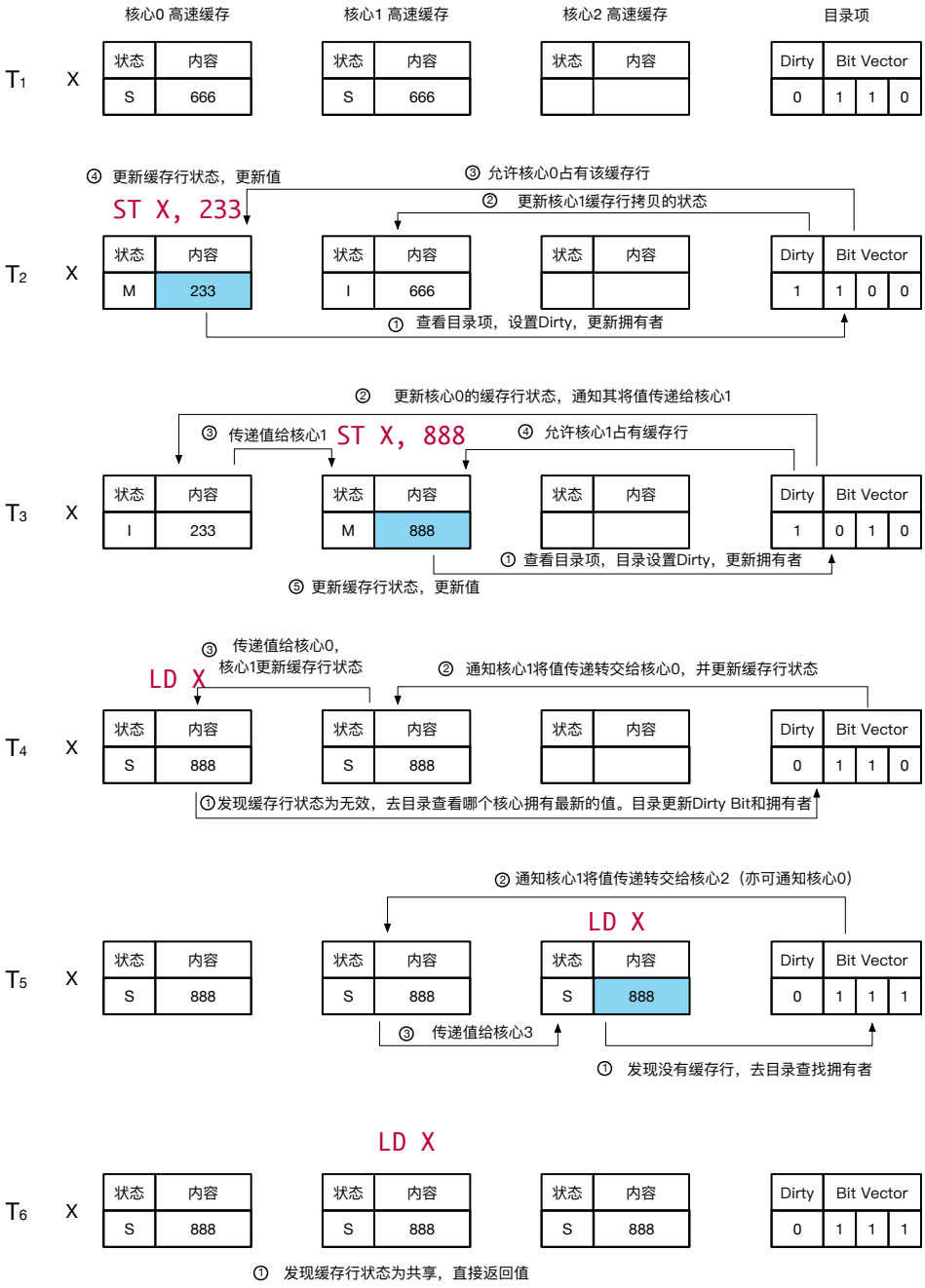

图 12.4: 目录式缓存一致性协议示例

12.1.3 系统软件视角下的缓存一致性

缓存一致性是硬件提供的、对上层软件透明的硬件特性。一些读者可能会疑惑："既然如此，为何系统软件设计者需要知晓这些硬件知识?"主要原因是系统软件的性能依赖于硬件实现，我们需要通过了解真实硬件实现，总结出其特性，才能针对这些特性设计出高性能系统软件。这些特性如下：

1. 多个核心对于同一缓存行的高频修改将会导致严重的性能开销。当多个核心需要修改同一缓存行时，需要缓存一致性协议来保证一致性。由于缓存一致性协议同一时刻只允许一个核心独占修改该缓存行，会造成多核执行流串行化，无法充分发挥出多核的性能优势；此外，多个核心对于同一缓存行的高频修改还会导致高速互联总线中产生大量缓存一致性流量，从而造成性能瓶颈。因此，系统软件开发者需要尽可能避免对单一缓存行的竞争。我们将在后续第12.4.1节以自旋互斥锁为例，介绍由于对锁变量所在缓存行的竞争导致的可扩展性问题，并介绍如何通过设计可扩展互斥锁来避免这个问题。

2. 伪共享（False Sharing）在多核中会造成严重的性能开销。伪共享是指本身无需在多核之间共享的内容被错误地划分到同一个缓存行中，并引起了多核环境下对于单一缓存行的竞争，从而导致无谓的性能开销。例如在软件中直接使用整型数组（如`int cnt[CORE_NUM]`）为每个核心提供一个独占的计数器，线程按照所在核心更新这些计数器。由于不同核心将更新不同的计数器，这些计数器本身不是共享的。但如果直接分配一个整型数组，这些计数器很可能落入到同一个缓存行中。当运行在不同核心的多个线程同时更新这些私有的计数器时，就会导致额外的缓存一致性开销。因此，系统软件开发者需要避免伪共享的发生，比如可将不需要共享的每核心本地数据分配到不同缓存行。

3. 多核环境下局部性同样重要。局部性包含时间局部性和空间局部性。时间局部性是指访问一个地址后程序在一段时间内还会访问相同的地址，而空间局部性则是指该地址相邻的内存将很可能会被访问。良好的局部性能够保证较高的缓存命中率，减少访问开销。而局部性较差的应用不仅会造成大量的缓存未命中，导致巨大的访存开销，还会逐出其他有用的缓存行，进一步影响性能。在多核环境中，任意核心的局部性问题都会导致共享的高速缓存（如 L2 Cache 与 LLC）受到影响，从而影响整个系统的性能，可谓牵一发而动全身。软件开发者需要注意让软件具有良好的局部性，高效使用高速缓存。

更多缓存一致性对于系统的性能以及可扩展性的影响，我们将在第12.4节讨论。

12.2 内存一致性与硬件内存屏障

本节主要知识点

- 有哪些常见的内存一致性模型？它们有哪些特性？
- 常见的架构中使用的是什么样的内存一致性模型？
- 在弱序一致性模型上编程，怎么保证访存操作可见顺序？
- 不同架构采用不同的内存一致性模型对于操作系统开发者来说意味着什么？
- 访存操作到底为何会出现乱序可见？在 *x86* 架构与 *ARM* 架构上又有什么不同？

12.2.1 多核中的访存乱序

现代处理器往往允许指令乱序执行。对于高时延的访存指令，处理器可以选择调度后续其他指令执行，从而隐藏访存操作的时延。然而，乱序执行意味着指令将不再按照程序顺序执行，访存操作的结果可能乱序地“全局可见”（即被所有核心观测到）[3]，最终导致执行结果不符合预期。

[3]注意，即使处理器不乱序执行访存操作，访存操作的结果也可能乱序地被其他核心观测到。我们将在第12.2.5节进行详细介绍。

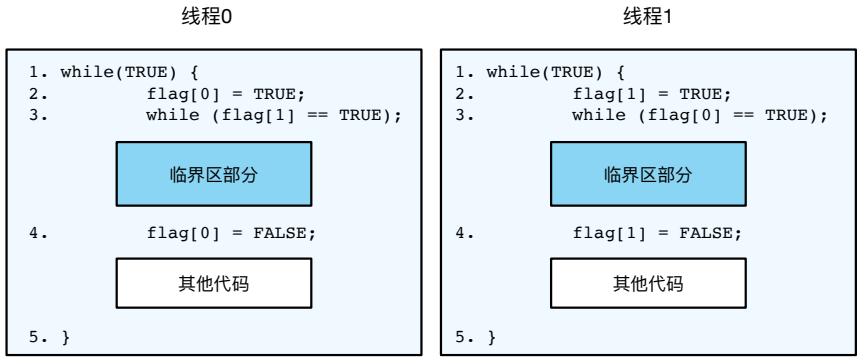

图 12.5: 皮特森算法前身：LockOne 算法

图12.5展示了LockOne算法，其是第八章介绍的皮特森算法的前身。这里使用LockOne算法是为了方便读者理解，访存乱序在皮特森算法中也会导致错误，其原因与LockOne算法完全一致。LockOne算法使用一组标记位flag来表示两个线程是否希望进入临界区。因此，在申请进入临界区时，线程将先设置自己的标记位，然后检查对方的标记位。如果对方同时想要进入临界区，则需要等待。在没有访存操作乱序的前提下，LockOne算法保证了互斥访问，也即两个线程不会同时进入临界区。需要注意的是，不同于皮特森算法，LockOne算法只保证了互斥访问，不保证**有限等待**与**空闲让进**。如果两个线程同时希望进入临界区，且在互相读对方的标记位之前，都已经设置了自己的标记位。此时，两个线程都不能进入临界区，陷入了无限等待。

在现代处理器上，由于会出现访存乱序的情况，因此LockOne算法的互斥访问无法保证。假设在线程 0 与线程 1 中，第 2 行的写操作与第 3 行的读操作发生了乱序，导致检查标记位的操作在自己的写标记全局可见（即对方的读操作可以读到其结果）之前发生。此时如果线程 0 与线程 1 同时申请进入临界区，它们乱序后的读操作可能读到对方的标记位均为false，导致两个线程同时进入临界区，打破了LockOne的互斥访问的保证。

12.2.2 内存一致性模型

内存一致性模型（Memory Consistency Model，简称为内存模型）明确定义了不同核心对于共享内存操作需要遵循的顺序。读写操作之间共有四类先后顺序需要保证，即读操作与读操作的顺序、读操作与写操作的顺序、写操作与读操作的顺序、写操作与写操作的顺序（为了简便，下文分别简写为读读、读

写、写读、写写）。针对同一地址或者有依赖关系[4]的访存操作，处理器可以保证其顺序。因此在不同的内存模型中，主要讨论的是针对**不同地址**以及**无依赖关系**的访存操作之间的顺序。下面本小节将按照从强到弱的顺序，依次介绍四种不同的内存模型。

严格一致性模型（Strict Consistency）

严格一致性模型是最严格的内存模型，也是最符合开发者直觉的内存模型。在严格一致性模型中，所有访存操作都是严格按照程序编写的顺序可见。此外，其要求所有核心对一个地址的任意读操作都能读到这个地址最近一次写的数据。因此，所有的线程看到的访存操作顺序都与其发生的时间顺序完全一致。在这种模型下实现的LockOne算法一定能保证互斥访问。但是，实现严格一致性模型要求使用全局一致的时钟[5] 以判定不同核心上执行的访存指令的时间先后顺序，增加了系统的实现难度。

顺序一致性模型（Sequential Consistency）

顺序一致性模型弱于严格一致性模型，其不要求操作按照其真实发生的时间顺序（即依据全局时钟定义的顺序）全局可见。顺序一致性模型提供了以下保证：首先，不同核心看到的访存操作顺序完全一致，这个顺序称为**全局顺序**；其次，在这个全局顺序中，每个核心自己的读写操作可见顺序必须与其程序顺序保持一致。不同于严格一致性模型，顺序一致性模型放松了对于时间顺序的要求，因此其中的读操作不一定能读到其他核上最新的修改。

图12.6展示了顺序一致性模型下运行LockOne算法可能的结果。这里使用局部变量A与B来分别存储读取的对方标记位的值。图中纵向箭头代表最终的全局顺序，横向箭头则表示访存操作在全局顺序中发生的时机。注意，这里的黑色箭头并非代表该访存操作实际发生的具体时间，仅仅代表在全局顺序中的先后顺序。图中列出来的 a、b、c 三种运行顺序都满足顺序一致性要求，因此对于任意一次执行，其最终全局可见顺序可能是 a、b、c 中的任意一种。此时，(A, B)的值有三种可能：(0, 1)，(1, 1)与(1, 1)。而对于图中的最后一种情况，由于其要求线程 1 写flag[1]的操作乱序到读flag[0]操作之后可见，违背了程序顺序，破坏了顺序一致性模型的要求，因此不可能出现。所以，顺序一致性模型能够保证LockOne算法的互斥访问。而如果使用严格一

[4]包括数据依赖、地址依赖等。我们将在第12.2.5详细介绍。

[5]严格一致性模型认为两个操作不可能在同一时刻发生。

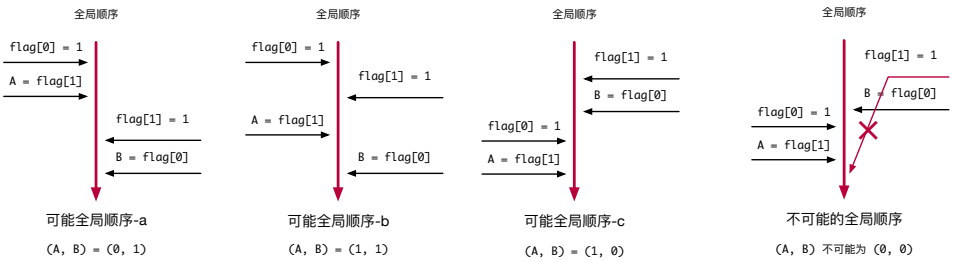

图 12.6: 顺序一致性模型下的 LockOne 算法

致性模型，则访存操作的可见顺序必须与执行顺序完全一致。因此对于任意一次执行，(A, B)的值都只有一种可能，其由访存操作的执行顺序决定。

TSO 一致性模型（Total Store Ordering）

为了达到更好的性能，TSO 一致性模型进一步弱化了访存一致性保证。在 TSO 一致性模型中，其保证对**不同地址且无依赖**的读读、读写、写写操作之间的全局可见顺序，只有写读的全局可见顺序**不能**得到保证。TSO 一致性模型通过加入一个写缓冲区达成优化性能的目的，该写缓冲区能够保证写操作按照顺序全局可见，TSO 一致性模型（Total Store Ordering）也因此得名。我们将在后续第12.2.5节进一步详细介绍。因此在LockOne算法中，TSO 一致性模型允许图12.6中最后一种情况发生，即写flag[1]操作乱序到读flag[0]操作之后可见，破坏了LockOne算法的互斥访问保证。

弱序一致性模型（Weak-ordering Consistency）

弱序一致性模型（后文简称为弱序一致性模型）提供了较 TSO 更弱的一致性保证。在一个核心上，弱序一致性模型不保证任何**不同地址且无依赖**的访存操作之间的顺序，也即读读，读写，写读与写写操作之间都可以乱序全局可见。这个特性导致在使用弱序一致性模型的设备上运行代码很容易出现违背开发者意图的情况。在弱序一致性模型中，LockOne算法同样不能保证互斥访问。除此之外，很多其他使用共享内存进行同步的程序也存在正确性问题。

代码片段12.1展示了一种基于共享内存的消息传递机制。两个需要通讯的线程分别运行proc_A与proc_B代码段。发送者先填充数据，再通过标记flag来通知接收者数据准备就绪（第 6 行）。为了保证消息被正确的传递，发送者需要保证写数据与写flag之间的顺序，而接收者需要保证读flag与读

```
1  int data = 0;
2  int flag = NOT_READY;
3  void proc_A(void)
4  {
5    data = 666;
6    flag = READY;
7  }
8
9  void proc_B(void)
10 {
11   while (flag != READY)
12     ; /* 循环忙等 */
13   handle(data);
14 }
```

代码片段 12.1: 基于共享内存的消息传递

数据之间的顺序。在 TSO 一致性模型中，写写与读读的顺序均可以得到保证，因此接收者能够读到正确的数据。而在弱序一致性模型中，由于写写与读读操作之间均允许乱序全局可见，在这种情况下，接收者可能读到错误的数据。

表12.1总结了本小节中介绍的四种内存模型对于不同地址、无依赖的访存操作可见顺序的保证。

表 12.1: 不同内存模型对于不同地址、无依赖访存操作可见顺序保证

	读读	读写	写读	写写
严格一致性模型	✓	✓	✓	✓
顺序一致性模型	✓	✓	✓	✓
TSO 一致性模型	✓	✓	×	✓
弱序一致性模型	×	×	×	×

12.2.3 内存屏障

不同于缓存一致性，内存模型对于上层软件**不是**透明的。在较弱的内存一致性模型中，为了保证特定访存操作的全局可见顺序，开发者必须手动添加**硬件内存屏障**（Barrier/Fence，简称内存屏障）。硬件内存屏障可以要求硬件保证访存操作之间的顺序。硬件往往提供多种不同的内存屏障指令来保证不同类型的访存操作的顺序。我们将在12.2.5节详细介绍内存屏障的工作原理。

代码片段12.2展示了通过添加内存屏障来保证数据data 能够正确地从

```
1  int data = 0;
2  int flag = NOT_READY;
3  void proc_A(void)
4  {
5    data = 666;
6    barrier();
7    flag = READY;
8  }
9
10 void proc_B(void)
11 {
12   while (flag != READY)
13     ; /* 循环忙等 */
14   barrier();
15   handle(data);
16 }
```

代码片段 12.2: 内存屏障示例

执行proc_A的线程传到执行proc_B的线程。这里假设使用了*弱序一致性模型*，即不保证无依赖的读写操作之间的顺序。因此在设置flag来告知数据就绪之前，需要保证data此时已经全局可见。这里使用一个内存屏障（第 6 行）保证写data与写flag之间的顺序，来完成这个目标。同样的，在proc_B中，读flag与读data的顺序也需要通过添加一个内存屏障来保证。而如果使用*TSO 一致性模型*，读读和写写乱序都是不允许的，因此不需要加入任何内存屏障。总的来说，不同的架构会根据自身使用的内存模型，提供不同的内存屏障指令。软件通过调用对应指令告诉硬件来保证特定类型访存操作之间的顺序。

12.2.4 常见架构使用的内存模型

严格的内存模型对开发者更加友好，处理器的行为更容易被开发者理解。但其会造成处理器设计复杂，导致制造成本高、处理器能效低。而较弱的内存模型硬件设计简单，能够挖掘更多的并行潜能。但开发者必须更加小心地添加硬件内存屏障来保证程序的正确性。除此之外，对于同步需求大的并行应用程序，在弱序一致性模型下频繁使用内存屏障会带来显著的性能开销。因此在多维度的权衡之下，不同的处理器制造商在其架构中使用了不同的内存模型。如 Intel 与 AMD 在 x86 架构下都使用了较强的 *TSO 一致性模型*。而 ARM 架构处理器则使用了*弱序一致性模型*。这个选择是根据 x86 与 ARM 应对的场景（桌面与移动）不同，考量处理器性能、功耗与成本等因素后确定的。表12.2展示了几种常见的体系结构使用的内存模型。

表 12.2: 几种常见架构使用的内存模型

	弱序一致性模型	TSO 一致性模型	顺序一致性模型
体系结构	ARM PowerPC	x86	Dual 386 MIPS R10000
使用场景	嵌入式，手机/平板， 高能效服务器	桌面电脑， 高性能服务器	被淘汰

在 x86 使用的 TSO 模型下，仅有写读操作会出现乱序。因此只需要在特定情况下添加内存屏障即可。而 ARM 处理器上的程序就没有这么幸运了。由于其使用的是弱序一致性模型，必须注意所有的无依赖的访存之间是否需要添加内存屏障保证可见顺序。在实际场景中，只有涉及到多核协同工作时，才需要考虑访存操作的可见顺序，使用合适的内存屏障。而需要同步的应用往往使用同步原语来保证正确性，其隐含了内存屏障的语义。在实现这些同步原语时需要根据不同的内存模型添加合适的内存屏障。此外，对于一些无锁的设计（如无锁队列），我们也需要考虑添加内存屏障来保证其正确性。

12.2.5 硬件视角下的内存模型与内存屏障🌶🌶

为了达到更好的性能，现代处理器往往允许访存操作乱序执行。在单个核心中，处理器使用了多种技术保证程序执行结果与按照程序顺序依次执行一致。

首先，处理器保证了有**依赖关系**的访存指令之间的顺序。比如，如果写操作的数据依赖于前序读操作的结果，则写操作必须等待读操作结束后才能开始执行。如`tmp = *ptr0; *ptr1 = tmp;`这两个操作中，写入`ptr1`地址中的值依赖于读`ptr0`的值的结果，则称这两个操作之间有**数据依赖关系**。除了有数据依赖，还有**地址依赖**（如写操作需要写的地址依赖于读操作读出来的值）、**控制依赖**（如写操作只有在读操作结束分支满足后才能执行[6]）。

除此之外，现代处理器还设计了**重排序缓冲区**（Re-Order Buffer, ROB），让指令按照程序顺序**退役**（Retire）。这里，退役对应顺序执行中的执行结束，其意味着该条指令对系统的影响终将全局可见。如果有的指令由于分支预测得到提前执行，但最终分支预测错误时，由于分支判断语句还未退役，这些指令不会提前退役，处理器就可以通过 ROB 追踪到这些被错误执行的指令，舍弃

[6]需要注意的是控制依赖不保证所有访存操作之间的顺序，需要结合具体硬件进行分析。

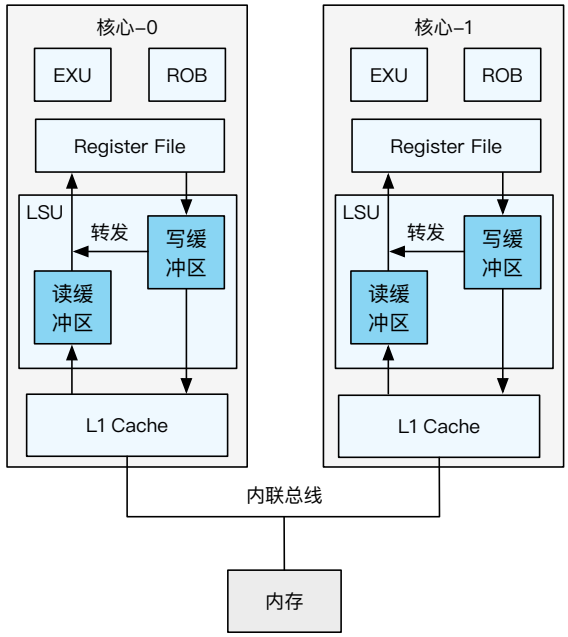

图 12.7: 现代处理器简化微体系结构示例

其执行结果并拿取正确的指令重新执行。因此，只在单核上运行的应用程序并不会受到影响，这些程序展现出的行为与严格按照程序顺序执行的行为相同。

然而在多核环境下，由于该系统中其它核心也可以观测到当前核心的运行结果，上述这些措施无法保证被其他核心观测的结果与严格按照程序顺序执行一致。其次，访存指令要完成在12.1节中介绍的缓存一致性流程后，才能顺利地被其它核心观测到。因此，如果访存指令等待缓存一致性流程结束后再退役，则会阻塞后续指令进入重排序缓冲区，导致性能受损。

现代处理器设计了另一套缓冲区：**读缓冲区**与**写缓冲区**，来解决这个问题。如图12.7所示，处理器在每个核心的**存取单元**（Load/Store Unit，简称为LSU）中预留了读缓冲区与写缓冲区。这个缓冲区用于暂存还没有满足缓存一致性的访存指令。在这种设计中，访存指令不再需要等待耗时的缓存一致性流程结束后再退役，而是放入对应的读缓冲区或写缓冲区后，就可以认为该指令已经完成。此时，访存指令只需等待前序的指令退役，且保证当前指令一定可以执行（如分支预测正确），便可以退役。但需要注意的是，此时访存指令退役并不代表其已经可以被其他核心观测到，其还在读写缓冲区中等待完成缓存一致性流程。退役只代表该指令一定可以执行，并一定在未来可以被其他核心观测到。这意味着，能否让其他核心按照程序顺序观测到该核心的执行结果的

重任落到了读缓冲区与写缓冲区身上。我们将一个访存操作完成缓存一致性流程、真正变得全局可见的过程称为**提交**（Commit）。提交与退役并不相等，一个访存操作需要等到其从重排序缓冲区退役后才会提交。

本节将详细介绍 x86 与 ARM 架构中内存模型的微体系结构实现，并介绍内存屏障指令如何保证访存操作可见顺序。注意，由于 x86 架构与 ARM 架构的手册都没有详细描述商用硬件的真正实现，因此本节描述的实现只是符合手册行为的一种实现可能，真实硬件中可能会选用不同的实现。

x86 架构下的 TSO 一致性模型

x86 架构采用了较强的 TSO 一致性模型。处理器中的读缓冲区与写缓冲区被用于维护不同访存操作全局可见的时机，因此 x86 架构的处理器中这两个缓冲区也被称为**内存顺序缓冲区**（Memory Ordering Buffer）。

我们首先介绍 x86 架构中的写缓冲区。写缓冲区用于缓存已经执行结束，但还没有变得全局可见的写操作。这里说的执行结束包括刚刚将指令提交到 LSU，还未通过重排序缓冲区退役的指令，以及已经退役但是还未能变得全局可见的写操作。我们这里规定，如果一个写操作满足了缓存一致性条件，真正离开写缓冲区到达 L1 cache，则代表该写操作的结果全局可见，称该写操作成功提交。对于任意写操作，离开写缓冲区需要满足以下几个前提条件：

1. 首先，该写操作必须要退役。一旦退役，当前核心将会认为这个写指令已经完成，该指令不可撤销。

2. 其次，该写操作所需的缓存一致性流程必须结束。如我们在第12.1节介绍的缓存一致性协议中，只有当前缓存行处于“独占修改”状态时，该写操作才能执行。

3. 最后，x86 架构的写缓冲区按照先入先出的顺序提交写操作。也即一个写操作必须等到前序写操作离开写缓冲区之后才能离开。因此，x86 架构使用的 TSO 模型中**写写**操作之间的顺序能够得到保证。

我们再来看读缓冲区。x86 处理器允许任意的读操作乱序执行。读操作并不存在提交的概念，但仍需要按照先入先出的顺序离开读缓冲区（即退役）。但是，如果某个在读缓冲区中的读操作还未退役，且此时该操作目标缓存行被其他核心修改，其会受到缓存一致性中的“失效”命令影响，舍弃当前读操作读到的结果，重新执行该读操作。所以在 TSO 架构中不会出现读读乱序（乱序执行的读操作读到的结果与顺序执行一致，否则会被要求重做）。因此，一个读操作的退役前提是：

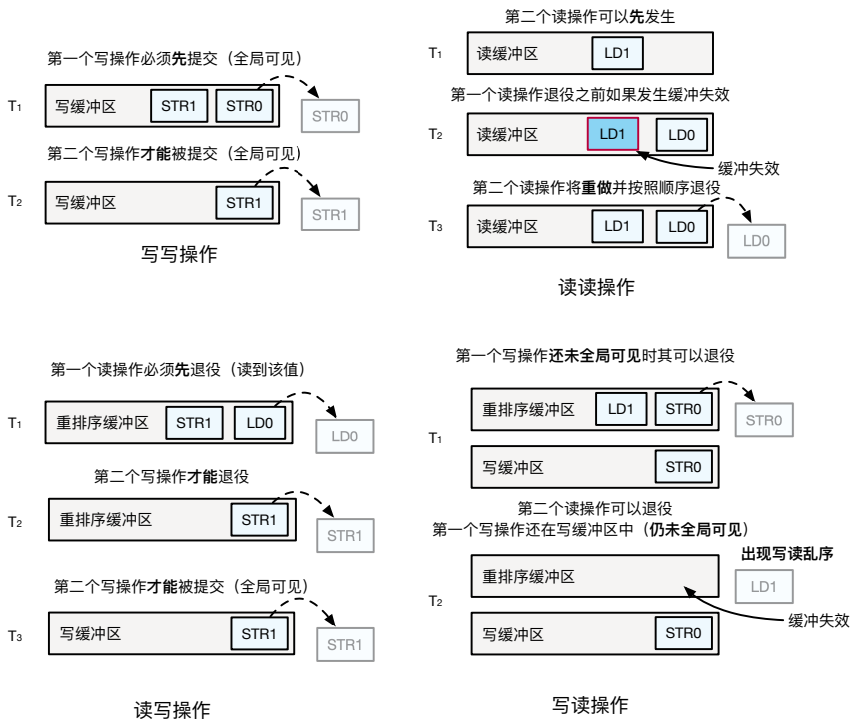

图 12.8: TSO 一致性模型中四类不同的操作组合行为

1. 首先，该读操作需要读取的值必须已经被读取到该核心。
2. 其次，程序顺序中该操作之前的所有操作必须已经退役。
3. 最后，读操作从读到值开始到退役之间没有收到目标缓存行“失效”的命令。若在此之间收到了目标缓存行“失效”的命令，则需要重新执行读操作。

在介绍完 x86 架构的读缓冲区和写缓冲区之后，下面通过图12.8展示了四种不同类型的访存操作组合在 TSO 一致性模型下的行为。其中STR0或LD0与STR1或LD1分别表示先后的两个访存操作。

写写。正如在介绍写缓冲区时讨论，由于写缓冲区保证了写操作会按照程序顺序提交，因此它们会按程序顺序全局可见，从而保证了**写写**顺序，即图中STR0与STR1将依照顺序全局可见。

读读。以代码片段12.1为例，假设乱序执行使得proc_B的读data操作（第 13 行，图中LD1）在读flag操作（第 11 行，图中LD0）之前发生，且读到的是初始数据0而非proc_A修改的666。而此时如果proc_B读flag操作读到

了proc_A修改的READY，那么由于proc_A可以保证写data到写READY的提交顺序(即保证**写写**的可见顺序),因此此时对data的写操作一定也已经提交完成。而根据缓存一致性协议，此时对于data缓存行的**缓存失效**通知一定已经到达了proc_B所在核心。由于读缓冲区按程序顺序退役，此时proc_B读flag操作刚刚完成，还未退役，因此读data缓存行操作也没有完成退役。所以，对于data缓存行的**缓存失效**通知会使得proc_B重做该读操作（图中 T_2 时刻），并读到修改值666。通过以上策略，x86 处理器就实现了对于**读读**操作的顺序保证。

读写。在 x86 处理器中，保证**读写**操作的顺序较为简单。由于重排序缓冲区已经保证了所有指令会按照程序顺序退役，而写操作的提交操作一定发生在退役之后，因此当一个写操作全局可见时（即提交），它之前的读操作一定已经完成并退役了（通过 ROB 保证，图中 T_2 时刻)。所以，TSO 架构能保证对于**读写**操作之间的顺序。

写读。最后对于**写读**操作，由于当写操作退役时，其值可能还没有变得全局可见（图中 T_1 时刻，STR0退役但还在写缓冲区中），但若其后续的读操作已经满足了退役的条件，造成读操作在写操作真正变得全局可见之前退役（图中 T_2 时刻)，最终破坏了写读之间的顺序。如果此时有使得读操作目标缓存行失效的命令到达，写读乱序将会被其他核心所观测到。

为了避免**写读**乱序，x86 处理器提供了mfence指令用于避免写读乱序。在我们上述的这种实现中，可以简单认为mfence指令会阻塞后续指令发送到 LSU，直到前序所有访存操作完成。以图 12.5中的LockOne算法为例，如果在两个线程的第 2-3 行之间添加了mfence指令，读对方flag的操作必须要等到写自己的flag操作提交之后才能发送到 LSU 并执行。因此能够保证LockOne算法的互斥访问。

ARM 架构下的弱序一致性模型

ARM 架构处理器选用了弱序一致性模型。弱序一致性模型不保证任何无依赖且针对不同地址的读写操作之间的顺序。这里需要注意，ARM 架构有一定的特殊性。虽然市面上很多处理器都使用了 ARM 架构，但是它们处理器内部实现可能会有明显差别。这是由于 ARM 对于处理器提供了很多可以由厂商决定如何实现的地方，其手册只定义了最终行为。因此，对于特定的处理器，其内部遵循的内存模型可能更加严格。同样，下面我们介绍的一种实现也只是遵循 ARM 手册的一种简化实现。

ARM 架构处理器虽然也有写缓冲区与读缓冲区，但与 Intel 的 x86 架构

处理器不同，这两个缓存不再提供顺序的保证。在 ARM 架构中，写缓冲区中的写操作可以不按照“先入先出”的顺序离开写缓冲区，当对应的写指令退役且目标缓存行缓存一致性流程结束后，该写操作便可以离开写缓冲区，变得全局可见。因此，对于任意的无依赖且针对不同地址的两个写操作，其全局可见的顺序不再受到约束。

ARM 架构下的读操作的退役机制与 x86 架构也存在很大差异。x86 架构下必须等到真正的值读入处理器内部（寄存器）且前序指令退役后才能退役。如果在等待过程中被缓存一致性协议标记为失效，则需要重新执行。因此，x86 架构能保证读读的顺序。而在 ARM 架构下，处理器可以在确保该读操作一定会发生（如分支预测成功）且前序指令退役时，就将该读操作退役。此时，这个读操作所需要读到的值可能还没有被读到处理器中。所以读操作虽然还是会按照程序顺序退役，但读操作发生的时机可能会违背程序顺序，造成**读读乱序**。

至于**读写**与**写读**这两种情况，其顺序更无法得到保证。这是由于读操作与写操作退役之后都不要求其真正执行完毕（需要读的值到达处理器或写的值全局可见），因此它们之间真实发生的顺序均没有严格要求，很可能会发生乱序。

如果需要保证 ARM 架构下访存操作之间的顺序，则有几种不同的选择。首先，类似 x86 架构，ARM 提供了一系列**硬件内存屏障指令**来满足不同场景的保序要求。其中最常用的一类内存屏障指令是`dmb`，其全称为 Data Memory Barrier。`dmb`指令能够保证其之前的访存指令与其之后的访存指令之间的顺序。`dmb`指令后还需要跟一个后缀用于表示其影响范围与影响的访存操作类型。对于影响范围，ARM 定义运行操作系统的处理器均属于`ish`，即 Inner Shareable Domain，因此在操作系统中一般使用`ish`后缀。对于影响的访存操作类型，ARM 提供了三种类型：一种是影响所有的访存操作，这种后缀无需增加任何新的关键字，如`dmb ish`；一种是保证`dmb`之前的写操作到其之后的写操作之间的顺序，这种后缀需要添加`st`关键字，如`dmb ishst`；最后一种是保证`dmb`之前的读操作到其之后的读/写操作之间的顺序，这种后缀需要添加`ld`关键字，如`dmb ishld`。处理器执行`dmb`指令时，会根据其后缀阻塞后续对应的访存指令发送到 LSU，直到前序对应的访存操作能够保证在指定的范围可见的**相对顺序**。这里的相对顺序是指前序指令并非已经全局可见，而是下层硬件能够保证同一核心后续的访存操作一定会在当前指令之后全局可见。除了`dmb`外，ARM 还提供了`dsb`、`ldar`、`stlr`等指令，本书不做过多介绍，有兴趣的读者可以参阅 ARM 手册 [4]。

除了硬件内存屏障，在 ARM 架构中还可以通过构造依赖关系来保证访存

操作的顺序。由于处理器执行指令时需要先等待该指令依赖的访存操作完成，所以存在数据依赖、地址依赖或控制依赖时，访存操作之间的顺序能够得到保证。

表 12.3: ARMv8 架构保序方案

	读读	读写	写读	写写
dmb/dsb ish	✓	✓	✓	✓
dmb/dsb ishld	✓	✓	×	×
dmb/dsb ishst	×	×	×	✓
ldar	✓	✓	×	×
stlr	×	✓	×	✓
数据依赖	×	✓	×	×
地址依赖	✓	✓	×	×
控制依赖	×	✓	×	×
控制依赖 + isb	✓	✓	×	×

表12.3展示了在 ARMv8 架构下不同的保序方案以及其适用的场景。其中 ✓ 表示能够保证，而 × 则代表不能保证。开发者需要根据具体使用场景选用合适的保序方案来保证程序的正确性。

12.3 非一致内存访问

本节主要知识点

- ❑ 什么是非一致内存访问（*NUMA*）架构？为何会有 *NUMA* 架构？
- ❑ *NUMA* 架构有哪些特性？

随着单处理器中核心数量增多以及多处理器系统的出现，单一的内存控制器逐渐成为了性能瓶颈。因此多核及多处理器系统将多个内存控制器分布在不同的核心或处理器上。这种设计在多处理器系统中非常常见，每个处理器往往被分配一个单独的内存控制器。因此，不同的核心有可以快速访问的本地内存，其也可以访问其他处理器上的远程内存，其访问时延远高于访问本地内存的时延。由于访问不同类型内存的时延不同，这种架构被称为**非一致内存访问**（Non-Uniform Memory Access，NUMA）。

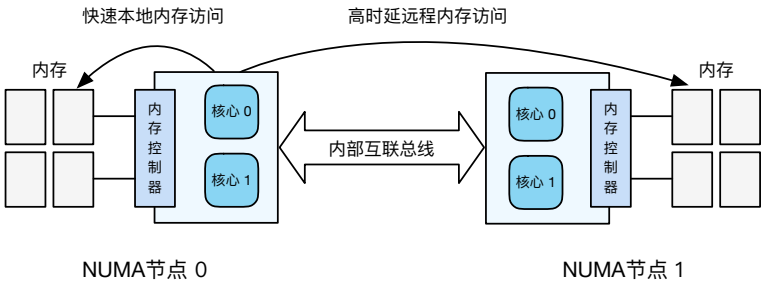

图 12.9: 非一致内存访问示例

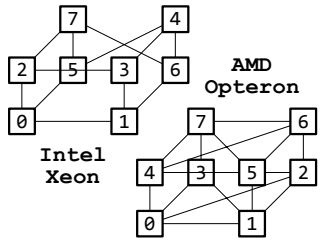

图 12.10: Intel 与 AMD 示例服务器拓扑结构图

图12.9是一个 NUMA 系统的示例，该系统包含两个 NUMA 节点。在这个示例系统中，一个处理器上的多个核心（如核心 0 与核心 1）访问内存的时延特性一致，因此划分到同一个 NUMA 节点（NUMA 节点 0）上。一个节点中的任意核心能够快速地访问本地的内存。一旦其需要访问远端其他节点的内存时，其需要通过**互联总线**（Interconnect）与远端节点通讯，造成了较高时延的远程内存访问。NUMA 架构有多种组成方式，一个 NUMA 节点可以是一个物理处理器，也可以是处理器中一部分核心。

现代单台服务器为了应对更高的计算需求，可以插入多个处理器，而每个处理器中同时拥有多个核心，因此其 NUMA 架构也更为复杂。图12.10展示了两个分别为 AMD 架构与 Intel 架构的示例服务器拓扑结构图 [10]。这两个服务器都拥有 8 个处理器插槽，分别对应着 8 个 NUMA 节点。由于节点数量众多，有些节点之间没有直接相连，请求需要通过两跳才能到达目标节点（如 Intel Xeon 与 AMD Opteron 服务器中的节点 0 与节点 7，均需要两跳）。因此访问时延差异也更加复杂。

表 12.4: 远程内存访问（cycles）时延特性

Inst.	0-hop	1-hop	2-hop
80 核心 Intel Xeon 服务器			
Load	117	271	372
Store	108	304	409
64 核心 AMD Opteron 服务器			
Load	228	419	498
Store	256	463	544

表12.4 [10] 展示了这两台服务器上访问本地以及远端内存的时延。表中数据的单位是**处理器周期**（Cycles），数字越大代表访问内存的时延越高。表格共有三列，分别代表访问本地的内存（0-hop），访问一跳的远程内存（1-hop，如 Intel 服务器中节点 0 上的处理器访问节点 1、2、5 上的内存），与访问两跳的远程内存（2-hop，如 Intel 服务器中节点 0 上的处理器访问节点 3、4、6、7 上的内存）。可以从表中看到，访问一跳的远程内存时延迟较本地的内存访问慢 2 倍以上，而访问两跳的远程内存访问更是达到了 3 倍以上。可见，如果应用所用内存随机分布在不同节点上，与只访问本地内存的应用相比，该应用将面临严重的远程内存访问开销。

表 12.5: 内存访问带宽 (MB/s)

Access	0-hop	1-hop	2-hop	Interleaved
80 核 Intel Xeon 服务器				
Sequential	3207	2455	2101	2333
Random	720	348	307	344
64 核 AMD Opteron 服务器				
Sequential	3241	2806/2406	1997	2509
Random	533	509/487	415	466

除了访问时延有所不同，远程内存访问的带宽上限也会受到内部互联总线的限制。表12.5 [10] 展示了这两台服务器本地及远端内存访问带宽的区别。由于内部互联总线的带宽限制，远端的内存访问的带宽较本地损失 40%。因此当应用需要访问大量远程内存时，访存带宽较访问本地内存更容易成为性能瓶颈。

NUMA 架构对于操作系统不是透明的，因此为了应对下层硬件的非一致

内存访问，操作系统应当分配、管理好硬件资源，尽可能避免应用中出现频繁的远程内存访问造成性能损耗。目前已经有很多研究工作提出了 NUMA 感知的设计方法，通过分配给应用本地的内存，避免使用远端的内存来降低 NUMA 对于应用性能的影响。更多细节我们将在第12.4.2节进行讨论。

12.4 操作系统性能可扩展性

本节主要知识点

- ❑ 为何软件会有可扩展性问题？其背后的原因是什么？
- ❑ 如何设计拥有良好可扩展性的软件？

上一节我们介绍了操作系统视角下的三种硬件特性，包括缓存一致性、内存一致性模型、非一致（时延）内存访问。本节将结合系统领域一些前沿研究，介绍系统软件的研究人员如何针对这些特性优化系统软件的性能，提升操作系统的性能可扩展性。什么是性能可扩展性？理想的可扩展性是随着核数增加 N 倍，软件的性能也能提升 N 倍。不过这明显是不切实际的。比如在现实中，如果有 10 个工人能够在 100 天造起一栋房子，如果将工人数量增加 100 倍、1000 倍，我们也无法将工期缩短致 1 天。在并行计算领域有阿姆达尔定律（Amdahl's Law） [3] 来描述并行计算的加速比。

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

公式中 S 描述加速比，p 为程序中可以并行的部分所占比例（$0 \leq p \leq 1$），而 s 为可以并行部分的加速比。在理想情况下，如我们如果有 N 个核，此时的并行部分加速比为 $s = N$。如果程序可以完全并行，即 $p = 1$，应用整体的加速比为理想的 $S = N$。而当 $p \neq 1$，应用的理想加速比会在有足够的核数下趋近 $S = \frac{1}{(1-p)}$。因此对于多核中运行的应用，如果本身算法中不可并行的部分 $1 - p$ 占比较大，添加核心数能带来的加速比便十分有限。此时需要优化应用本身逻辑与算法，以提升应用的可扩展性。除了算法本身导致的可扩展性问题，上一节介绍的多核多处理器硬件特性同样也会影响应用的可扩展性，甚至出现应用在多个核心上运行的性能不及单个核心的情况。下面将分别介绍在多核环境下由于对单一缓存行竞争与 NUMA 架构下由于频繁远程内存访问导致的可扩展性问题，并从同步原语角度介绍如何改善这两种场景下的性能可扩展性。

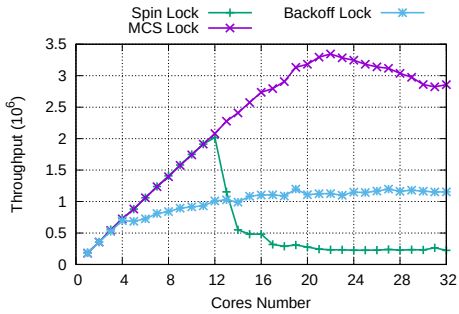

图 12.11: 华为泰山服务器（鲲鹏 916）互斥锁可扩展性测试

```
1 while (1) {
2   lock(global_lock);
3   global_cnt++;
4   random_access_cacheline(global_cacheline);
5   unlock(global_lock);
6   nops();
7 }
```

代码片段 12.3: 微基准测试

12.4.1 单一缓存行高度竞争导致的可扩展性问题

在多核环境中，应用不能并行的部分需要使用我们在第八章中介绍的同步原语来保证程序的正确性。但是，这些同步原语往往会造成可扩展性问题。图12.11展示了在华为 ARM 架构服务器鲲鹏 916 上使用互斥锁面临的可扩展性问题。在这个测试中，我们在每个核心上运行一个线程。代码段12.3展示了微基准测试中一个线程的执行流程。该线程将不断执行一个循环。在循环中，该线程将竞争一个全局的互斥锁`global_lock`。获得锁后，该线程在临界区中将更新全局的计数器`global_cnt`，然后读取并修改 1 个全局缓存行`global_cacheline`用于模拟应用程序临界区对于全局共享变量的修改。而两个临界区之间，线程将执行一定数量的空操作（`nops`），用来模拟非临界区代码。图中的 x 轴从左至右增加并行执行的核心数，而 y 轴则代表对应核心数下测试程序的吞吐率。从图中可以发现，当增加到 12 核时，使用自旋锁（Spin Lock，在8.1.4节介绍）的测试程序性能达到峰值。一旦加入更多的核数，其性能不仅不会上升，反而会出现断崖式下坠。

系统领域的研究者通过对互斥锁性能建模，详细分析了这个问题 [5]。研究者发现，当执行核心数超过一定阈值后，就会出现同一时刻有多个核心在等

待获取互斥锁的情况，且等待的线程数量会随着核心数的进一步增多而增加。但是在理想情况下，即使更多核心同时竞争互斥锁也只应导致性能无法继续提升，并不会损害性能，更不会出现如图12.11展示的断崖式下坠。性能断崖式下坠的现象实际上是由于在第12.1节介绍的缓存一致性导致的。由于自旋锁（代码片段8.7）是通过修改全局单一变量`*lock`来代表获取以及释放锁，因此自旋锁的获取与释放操作均会造成对单一缓存行拥有权的竞争（同一时刻，只有一个核心能够“独占修改”这一个缓存行）。而当多个核心对同一缓存行进行高频的访问与修改时，缓存一致性开销十分巨大。这是由于自旋锁在等待期间会不断地访问并尝试修改`*lock`所在缓存行。当同一时刻竞争核数增多时，缓存行的状态与拥有者也不断改变，最终消耗大量时间在缓存一致性协议上，导致互斥锁无法快速有效地在不同的核心之间传递。此时，锁在竞争者之间传递的开销（主要是缓存一致性开销）伴随着竞争者数量增加而增加。

既然断崖下坠是由于同一时刻有太多竞争者在抢`*lock`所在的缓存行，减少这些竞争者数量就成为了解决可扩展性问题的关键。我们首先介绍一种直观的、无需大量修改的策略：**回退策略**（Back-Off）。

回退策略的核心思想非常简单，当竞争者拿不到锁时，它就不再继续尝试修改该缓存行，而是选择等一段时间再去拿锁。为了避免多个竞争者的等待时间相同导致的再次修改时依旧出现竞争，回退策略为竞争者设定了不同的等待时间。比如等待随机时长或依照一定序列依次加长等待时间。图12.11中展示了使用指数回退策略的 **回退锁**（Back-Off Lock）在华为泰山服务器（鲲鹏916）上微基准测试的性能。该锁在竞争获取失败时，以指数时间回退。可以发现在少于 14 个核心时，回退锁的性能弱于自旋锁。这是由于回退锁在竞争程度不高时，就已经开始采用回退策略。此时会出现一些竞争者睡眠时锁被释放，且无其他竞争者获取锁的情况，从而导致一定的性能开销。但当有更多核心时，其性能超过自旋锁，且随着核心数增加，回退锁的吞吐率十分稳定，不再出现可扩展性断崖。

回退策略虽然缓解了对于缓存行的竞争，但并没有解决本质问题，只是一定程度减少了问题的出现。而良好的可扩展互斥锁需要保证其竞争开销（如缓存行失效的次数）不应随着竞争者数量增多而加大。下面将通过介绍可扩展的队列锁：MCS 锁，分析如何设计具有良好性能可扩展性的系统软件。

MCS 锁

MCS 锁 [8] 是由 John M. Mellor-Crummey 与 Michael L. Scott 在 1991 年提出，MCS 锁也因此得名（MC 与 S 分别为两位作者姓氏首字母）。同排号

```
1 void *XCHG(void **addr, void *new_value)
2 {
3   void *tmp = *addr;
4   *addr = new_value;
5   return tmp;
6 }
```

代码片段 12.4: XCHG 操作示意

锁（读者可以回顾第8.1.4节）类似，MCS 锁拥有一个等待队列。为了避免对单一缓存行的竞争，MCS 为每一个竞争者都准备了一个节点，并插入到一个链表中。这样锁的持有者可以通过链表找到下一任竞争者并将锁传递。因此竞争者只需等待在自己的节点上，由前任锁的持有者通过修改自己节点上的标记来完成锁的传递，无需像排号锁一样通过竞争全局缓存行的方式来检查是否轮到自己。下面详细介绍 MCS 的加锁和放锁操作流程。

为了实现 MCS 队列锁，我们这里引入一个新的原子操作：`atomic_XCHG`。`XCHG`的操作如代码片段12.4所示，它会将`addr`地址上的值修改为新值`new_value` 并返回该地址上原来的值。而`atomic_XCHG`则会保证交换操作的原子性。

代码片段12.5展示了 MCS 锁的实现。该实现省去了内存屏障，要求访存操作严格按照程序顺序可见。MCS 锁的元数据中记录着等待队列的队尾指针`tail`。当锁为释放状态时，该指针为空。一旦有竞争者出现，该指针将指向等待队列的队尾。每一个锁的竞争者都需要创建自己的 MCS 节点`MCS_node`。该节点包括一个标志位`flag`，它可以表示两种状态：`WAITING`，表示当前节点对应的竞争者应当等待；`GRANTED`，表示当前节点对应的竞争者被授权可以进入临界区。此外，每个节点还有一个指向等待队列下一个节点的`next`指针。注意，为了保证每个节点都在不同缓存行上，避免出现在12.1中提到的**伪共享**的问题，这里要求`MCS_node`依照缓存行大小对齐。不同于之前的自旋锁，MCS 锁的`lock`与`unlock`操作除了需要传入锁的结构体，还需要传入当前线程的 MCS 节点`me`。

当一个线程调用`lock`尝试获取 MCS 锁时，其先初始化自己的等待队列节点（第 13、14 行）。其后，该线程利用`atomic_XCHG`操作将队尾指针`lock->tail`交换为指向自己的指针，并将原来的队尾指针的值存入`tail`（第 15 行）。若原来的指针为空，代表该锁为空闲状态，此时该线程可以直接进入临界区执行。否则，该线程将链入自己的节点（第 17 行）并等待在自己的 MCS 节点上（第 18 行）。

```
1  struct MCS_node {
2    volatile int flag;
3    volatile struct MCS_node *next;
4  } __attribute__((aligned(CACHELINE_SZ)));
5
6  struct MCS_lock {
7    struct MCS_node *tail;
8  };
9
10 void lock(struct MCS_lock *lock, struct MCS_node *me)
11 {
12   struct MCS_node *tail = 0;
13   me->next = NULL;
14   me->flag = WAITING;
15   tail = atomic_XCHG(&lock->tail, me);
16   if (tail) {
17     tail->next = me;
18     while (me->flag != GRANTED)
19       ;
20   }
21 }
22
23 void unlock(struct MCS_lock *lock,
24             struct MCS_node *me)
25 {
26   if (!me->next) {
27     if (atomic_CAS(&lock->tail, me, 0) == me)
28       return;
29     while (!me->next)
30       ;
31   }
32   me->next->flag = GRANTED;
33 }
```

代码片段 12.5: MCS 队列锁实现

当持有锁的线程通过unlock释放锁时，将先检查等待队列中是否还有其他线程等待在该锁上（第 26 行）。如果有其他线程等待，则依照链表顺序，通过修改后序等待者节点中的标记位为GRANTED来通知该节点进入临界区（第 32 行）。若当前线程已经是等待队列里的最后一个，该线程则需要原子地将等待队列的队尾指针置为空，表示该锁已经释放（第 27 行）。如果原子操作失败，说明此时有新的线程已经交换了队尾指针，该线程需要等待新竞争者链入并将锁传递给新的线程（第 29 行）。

图12.12结合一个实际例子，展示了获取 MCS 锁与释放 MCS 锁的流程。

- 在 T_1 时刻，等待队列中一个有四个节点。此时，处于队首的竞争者拥有锁，其标志位为GRANTED。

- 在 T_2 时刻，有一个新的竞争者出现。其首先初始化自己的节点：将标记位填为WAITING，并将next指针置为空。接着，该竞争者将原子交换元数据中的队尾指针指向自己。其发现，之前队尾指针不为空，而是指向原等待队列的队尾，说明该锁正被其它竞争者持有。因此，该竞争者修改原队尾的next指针指向自己，从而链入等待队列，并等待前序节点将锁传递给自己。

- 在 T_3 时刻，持有锁的第一个线程希望释放锁。其发现等待队列中存在其他的等待者，因此通过修改后序等待者的标记位为GRANTED来通知其进入临界区，完成锁的传递。之后，该线程可以回收自己的节点，或者留到下次继续使用。

- 最后在 T_n 时刻，终于轮到等待队列最后一个线程放锁时，不巧又有一个新的线程希望得到锁（MCS 节点 6）。由于新线程还没成功交换尾指针，原队尾线程并不知晓新竞争者的存在。作为队尾节点，其需要通过原子的 CAS 操作，判断队尾指针是否指向自己，如果指向自己则将其换为空。由于新的竞争者的存在，其也在尝试交换队尾指针，因此原队尾线程的 CAS 操作可能会失败。当原子 CAS 失败，意味着有新的线程已经成功交换了队尾指针。此时只需要等待新的线程通过更新原队尾的next指针链入等待队列，即可按照与 T_3 时刻相同的流程将锁传递下去。而若原子的 CAS 成功，此时意味着锁已经被成功释放。新的竞争者成功交换队尾指针时，会发现队尾指针为空。因此其可以直接进入临界区。

下面分析为何这样设计能够为 MCS 锁带来更好的可扩展性。如果一个锁要具有良好的可扩展性，锁竞争的开销（如缓存行失效的次数）不应随着同一

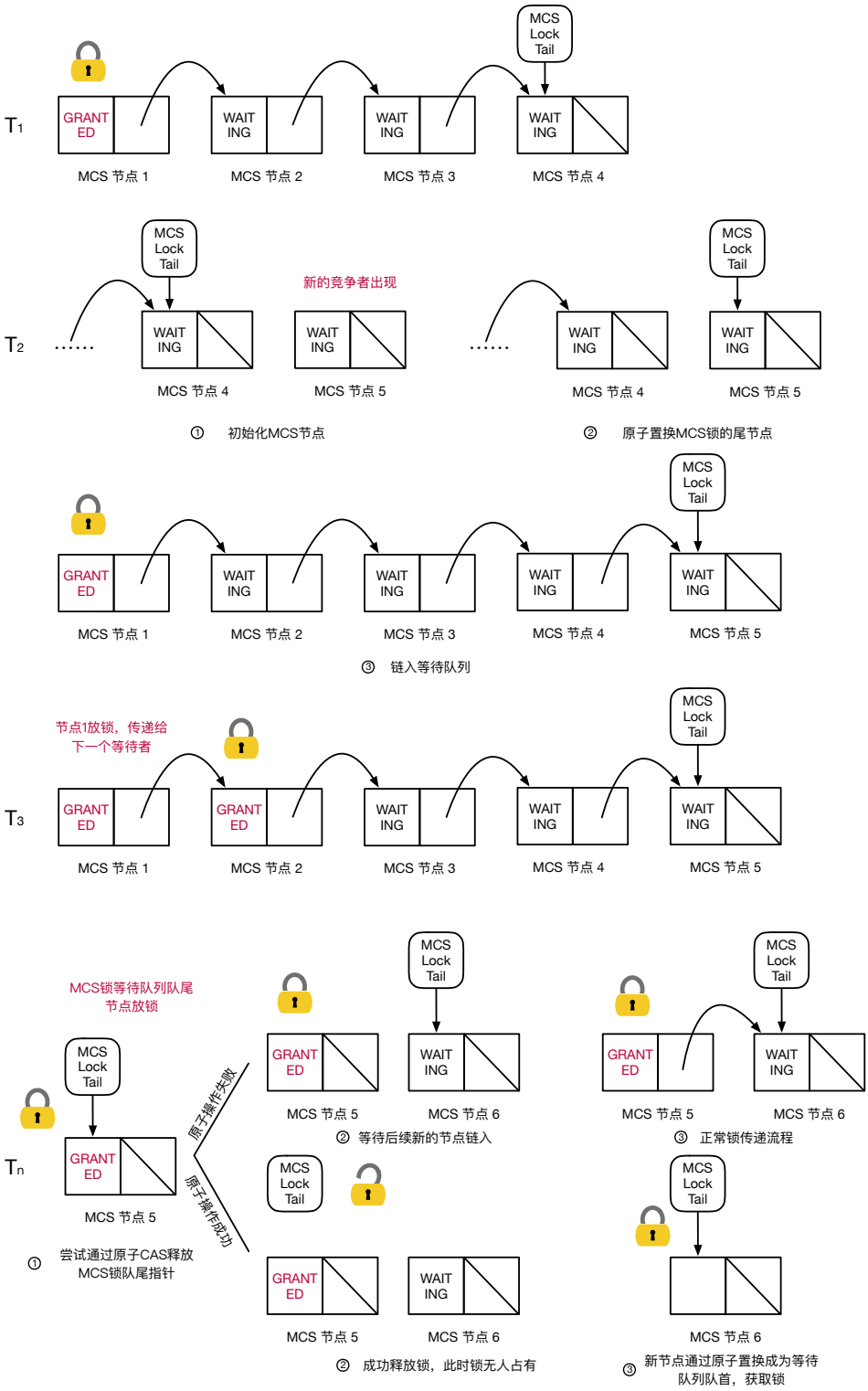

图 12.12: MCS 操作示例

时刻竞争者数量增多而加大。在自旋锁中，随着竞争者数量增多，单一缓存行的竞争加剧，最终导致关键路径上平均每次获取锁造成的缓存失效数量急剧上升。而对 MCS 锁而言，当同一时刻竞争者数量增多时，理想情况下关键路径上锁传递都只涉及两次缓存失效：分别为被继任者修改的`me->next`所在缓存行与继任者初始化的`me->next->flag`所在缓存行。当然，对于`tail`的竞争也有可能暴露在关键路径上，但当参与竞争的线程较多时，等待队列中一般都已经链入了多个竞争者，因此锁可以快速地在等待队列中传递，在关键路径上不会涉及对于`tail`的修改。图12.11的微基准测试中，MCS 锁展现出良好的性能可扩展性。可以看到在核心数增多时，MCS 锁的性能不会出现大幅下降。

小思考

MCS 锁性能一定比自旋锁好？我们应该怎么样在这两个锁之间选择？

自旋锁获取锁与释放锁所需的操作均非常精简，在只有少数核心竞争时，其性能较 MCS 锁更好。因此这两个锁并不存在好坏之分，需要依据具体场景下的竞争程度来判断到底选用哪一种锁。操作系统中很多问题都与其类似，很难找到一个能应对所有场景的最优解决方案，很多情况下需要对具体场景进行具体分析。系统研究者也想办法结合多种方案之间的优点，设计自动依据具体场景特征切换策略的解决方案。比如在 Linux Kernel 的 3.15 版本中，就引入了 qspinlock（queue spinlock）这一种新的锁。这种锁结合了自旋锁与 MCS 队列锁的优势，在不同竞争程度下使用不同的策略。qspinlock 拥有快速路径与慢速路径。当一个线程尝试获取该锁时，其首先尝试快速路径，通过原子的 CAS 获取该锁。一旦获取失败，该线程将采取类似 MCS 锁的机制，将自己加入等待队列。这种设计能适应竞争程度变化的复杂场景。当竞争程度低时，线程可以走快速路径快速地拿到锁，而当竞争程度高时，类似 MCS 锁的设计则能提供良好的性能可扩展性。

12.4.2 NUMA 架构中频繁远程内存访问导致的可扩展性问题

本节主要知识点

- *NUMA* 架构为可扩展性带来什么样的新挑战？
- 为何 *MCS* 锁在 *NUMA* 架构下会出现可扩展性问题？
- 我们该如何设计才能在 *NUMA* 架构下拥有良好的可扩展性？

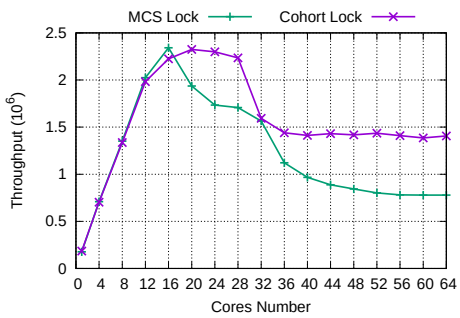

图 12.13: 华为泰山服务器（鲲鹏 916）互斥锁 NUMA 环境可扩展性测试

上一节我们分析了由于对于单一缓存行的竞争导致的互斥锁可扩展性问题。然而，互斥锁仅仅是程序中的一部分，其保护的**共享资源**（如全局变量）本身同样可能造成可扩展性问题。比如，两个不同核上的线程在临界区内访问了同一个缓存行。当锁在这两个线程之间传递时，该缓存行也需要在这两个不同的核之间传递，造成一定的通信开销。这个通讯开销在 NUMA 架构[7]中会更加的显著。假设这两个线程分别运行在不同的 NUMA 节点上，该缓存行就会涉及耗时的跨 NUMA 节点的缓存一致性通信。此外，一旦缓存行被赶出高速缓存，需要重新从内存读取时，访问远程内存会带来更加严重的性能开销，造成严重的可扩展性问题。在 NUMA 环境下，无论是使用之前介绍的排号锁亦或是刚介绍的 MCS 队列锁，锁都会按照申请获取锁的顺序依次传递。如果来自不同 NUMA 节点的竞争者依次申请获取锁，那么锁的元数据以及临界区内共享数据所在的缓存行都将在不同的 NUMA 节点之间传递，这将会非常耗时。

为了进一步验证这个结论，下文使用华为泰山服务器（鲲鹏 916）进行了一系列测试。该服务器一共有 2 个处理器插槽，每个插槽上插了一颗有 32 个核心的 ARM 架构处理器。每颗处理器中又进一步分成了 2 个 NUMA 节点，因此整个系统一共有 4 个 NUMA 节点，且每个节点有 16 个核心。图12.13展示了在该服务器上使用互斥锁在 NUMA 环境中面临的问题。相较于前一小节的微基准测试，这里稍有不同。如代码段12.6所示，这里临界区将访问并修改 *10* 个共享缓存行中的数据，而非之前的 *1* 个。另外，测试使用的核心数进一步从 32 核扩展至 64 核。可以从图12.13中看到，12.4.1节中介绍的 MCS 锁依旧会面临严重的可扩展性问题。在核数上升到 16 个核心时，其性能会开始逐步下降。对比图12.11中的结果，这里出现的性能下降主要是由于线程临界区访问了更多的缓存行，而当核心数超过 16 个核心时，锁的竞争者就可能出现在不

[7]主要为拥有缓存一致性保证的 NUMA 架构（Cache Coherence-NUMA, CC-NUMA)。

```
1 while (1) {
2   lock(global_lock);
3   global_cnt ++;
4   for (i = 0; i < 10; i++)
5     random_access_cacheline(global_cacheline[i]);
6   unlock(global_lock);
7   nops();
8 }
```

代码片段 12.6: NUMA 环境下的微基准测试

同的 NUMA 节点。当锁持有者在不同节点上时，临界区内访问的缓存行也需要在不同的 NUMA 节点之间迁移，导致巨大的开销。

为了解决这个问题，研究者提供了一系列 NUMA 感知（NUMA-aware）的设计，用于增强访存操作的局部性，减少发生跨 NUMA 节点的缓存一致性。这类 NUMA 感知设计的核心，是在保证正确性的同时尽可能保证一段时间内访问都能在本地命中（包括本地的缓存行和内存），从而避免在关键路径上出现太多远程内存访问（包括通过缓存一致性访问在其他节点高速缓存中的缓存行）。为了进一步说明这种设计方针，这里利用 Cohort 锁 [6] 作为一个详细的例子分析如何设计 NUMA 感知的应用。该锁是由 Dave Dice 在 2012 年提出，其展现了在 NUMA 环境下优化可扩展性的基本思想。

Cohort 锁的设计核心是在一段时间内限制互斥锁在单个 NUMA 节点内部传递，从而在关键路径上剔除耗时的跨节点缓存一致性。为了达到这个目的，Cohort 锁采取了两层锁的设计，第一层是唯一的全局锁，第二层是对应每个 NUMA 节点的本地锁。当某一个 NUMA 节点上的核心持有 Cohort 锁时，其需要同时持有全局锁以及本地锁。而当该核心执行解锁操作时，如果本地节点有其他竞争者，则该核心不会释放全局锁，而是释放本地锁并将全局锁的拥有权转给下一个本节点的竞争者，从而保证 Cohort 锁在一段时间内在本地的线程之间传递。

图12.14展示了在 3 个 NUMA 节点的系统中 Cohort 锁传递的示例。最初，NUMA 节点 0 在核心 0 上的竞争者（记为 0-0）持有 Cohort 锁，即同时持有节点 0 的本地锁以及全局锁。当其释放锁时，其将只释放本地锁，将其传递给下一个竞争者 0-1。在一段时间内，只有 NUMA 节点 0 内的竞争者可以获取锁。当所有的本地竞争者都执行完自己的临界区之后，本节点最后的持有者将释放本地锁与全局锁，将全局锁传递给节点 1。而节点 0 上的后序竞争者都需要重新获取全局锁才能进入临界区。

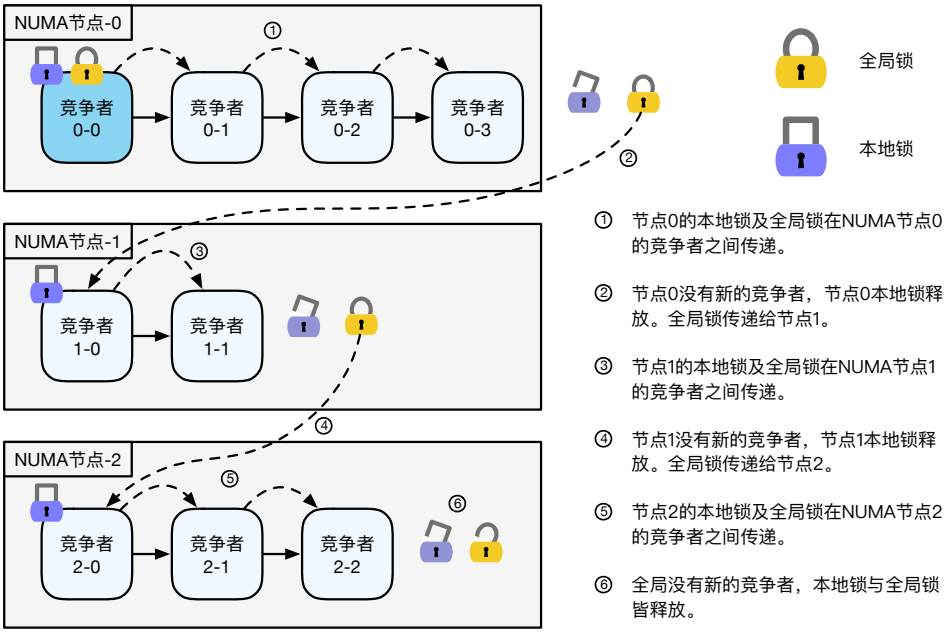

图 12.14: Cohort 锁传递示例

```
1  struct lock *global_lock;
2  struct lock *NUMA_lock[NUMA_NUM];
3
4  void routine(void)
5  {
6    int my_node_id = get_running_node_id();
7    if (lock(NUMA_lock[my_node_id]) == NO_WAITER)
8      lock(global_lock);
9    // 临界区
10   if (unlock(NUMA_lock[my_node_id]) == NO_WAITER)
11     unlock(global_lock);
12 }
```

代码片段 12.7: 使用 Cohort 锁示例

代码片段12.7展示了 Cohort 锁的执行流程。这里稍微修改了lock与unlock接口。这两个操作将返回是否存在其他等待者。在加锁阶段，线程将先获取本地的锁（第 7 行）。如果已经有本地的等待者（返回值不等于NO_WAITER），其只需要等待前序持有者将锁传递给自己即可进入临界区（第 9 行）。而如果没有等待者（返回值等于NO_WAITER），则该线程需要代表该节点去获取全局锁（第 8 行）。同理，在释放锁的时候，如果有后序的等待者，则不释放全局锁（第 10 行）。全局锁的所有权自然传递给本地的下一个持有者。而如果没有后序等待者，则需要释放全局锁（第 11 行），让其他节点有能力进入临界区。

我们可以从图12.13中看到：相较于 MCS 锁，Cohort 锁在 NUMA 环境下拥有更好的可扩展性。不过，Cohort 锁在超过 32 个核心、出现跨处理器插槽的缓存一致性通信时，其性能也会显著地下降。这是由于虽然 Cohort 锁在一段时间将竞争限制到了本地，其仍然无法避免开销巨大的跨节点缓存一致性。可见，跨节点缓存一致性开销会对应用性能带来严重的影响。

相较于 MCS 锁，Cohort 锁牺牲了一定的公平性。MCS 锁遵循先入先出的规则来传递锁，而 Cohort 锁则优先传给本地的竞争者。在极端情况下甚至可能造成其它节点的竞争者饥饿。因此 Cohort 锁在使用时还会规定锁在本地传递次数的上限，从而保证一段时间内的公平性。

我们来总结一下 Cohort 锁的核心设计思路。Cohort 锁为了避免跨 NUMA 节点的缓存一致性开销，在一段时间内将竞争限制在本地。Cohort 锁不仅保证了锁的元数据所在缓存行在一段时间内会留在本地，也保证了临界区内共享数据所在缓存行不会频繁地在不同 NUMA 节点间迁移。

除了 Cohort 锁的方法以外，研究者还提出了代理执行的方案来将竞争限制在本地，即代理锁（Delegation Lock） [7]。代理锁将所有的临界区全部收集到一个核心上，并只在这一个核心上执行，从而最大程度地减少缓存行失效导致的跨节点的缓存一致性。需要执行的临界区以函数指针及参数的方式发送给这一个核心，在临界区执行完毕后再由该核心发送回临界区返回值。由于这里发送执行临界区请求的过程不在关键路径上，极大地提升了性能。无论是代理锁还是 Cohort 锁，其最终目标都是通过 NUMA 感知的设计，尽可能避免共享数据产生跨节点竞争。

小思考

从操作系统的角度出发，我们如何提供 NUMA 感知的能力？

首先我们需要为上层应用提供 NUMA 感知的内存分配接口，让上层应用

显式地指定分配内存所在位置。其次，对于没有使用这些接口的应用，操作系统需要尽可能地将内存分配在本地，避免远程内存访问。最后，操作系统调度时需要尽可能避免跨 NUMA 节点的线程迁移。

12.5 案例分析: Linux 内核中的 NUMA 感知设计

为了进一步介绍操作系统如何提供 NUMA 感知的支持，这里以 Linux 内核中 NUMA 感知的内存管理 [1] 以及 NUMA 感知的调度 [2] 为例进行详细分析。

12.5.1 NUMA 感知的内存管理

针对 NUMA 环境，Linux 内核提供了多种**内存分配策略**以及**内存分配模式**，以便在 NUMA 环境中合理地分配内存。

其中内存分配策略指定了内存分配时，应该遵循谁制定的规则（即内存分配模式）。这些策略包含了：由任务进程自己指定自己分配内存时采用的模式（即任务指定策略）；以及直接给定特定虚拟地址空间，在该空间内进行内存分配时采用的模式（即虚拟地址空间指定策略）等。用户可以分别使用内核提供的`set_mempolicy`与`mbind` 这两个系统调用来实现上述功能。若用户没有选择内存分配策略，则操作系统内核将接过决定分配模式的大旗，帮助进程选择合适的分配模式。

而至于内存分配模式，Linux 内核提供了绑定模式、优先模式与交错模式。绑定模式顾名思义，其将从指定的 NUMA 节点（通过上一段介绍的系统调用指定）上分配内存。应用可以使用该模式选择在自己运行的 NUMA 节点上分配本地内存。而优先模式则在分配失败时，尝试从指定节点最近的节点上分配内存。交错模式则会从给定的节点中以页为粒度交错地分配内存。如果应用没有选用任何模式，操作系统则会采用优先模式帮助应用在其运行的节点及邻近的节点上分配内存。

总而言之，Linux 内核从两方面针对 NUMA 环境下的内存分配进行了优化。首先，Linux 内核提供了系统调用，让应用能够根据实际需求指定特定节点进行内存分配；而对于没有使用这些接口的应用，内核则尽可能在应用运行的节点上分配本地的内存。

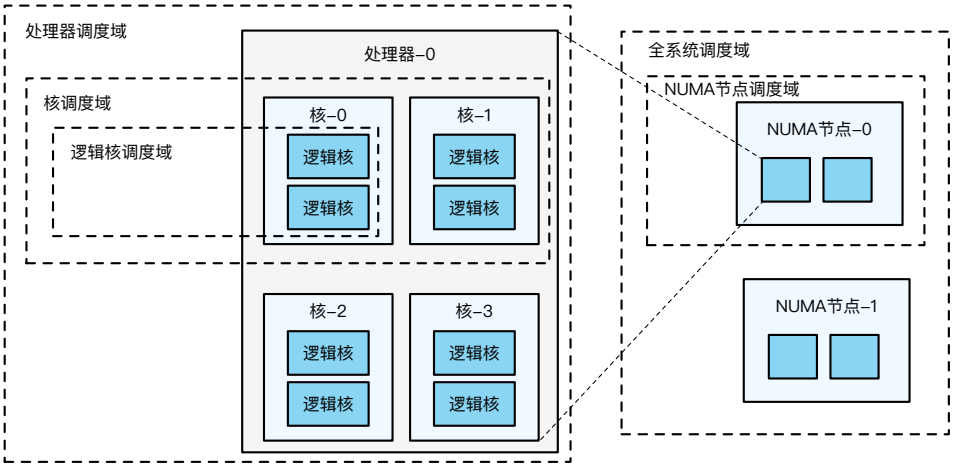

图 12.15: 调度域示例

12.5.2 NUMA 感知的调度

Linux 内核中的调度策略也考虑了 NUMA 环境。对于多核多处理器硬件，Linux 内核中的调度器主要需要权衡两个问题：一方面需要尽可能让任务均匀地分布在系统的各个核心上，达到更好的**负载均衡**；另一方面，频繁地在不同的核心之间迁移任务意味着丧失了任务的本地性，特别是在 NUMA 环境下，跨 NUMA 节点的任务迁移将导致巨大的迁移开销。因此 Linux 内核引入了**调度域**（Scheduling Domain）的概念，将 CPU 分成了不同的调度域。每个调度域是具有相同属性的一组 CPU 的集合，并根据硬件特征划分成不同的层级，呈现出一种树状结构。

如图12.15展示了一个 NUMA 系统的调度域。最下层是逻辑核调度域，每个域包含了同一个物理核中的逻辑核，这些逻辑核共享高速缓存，因此在它们之间迁移的开销最低；向上一层是核调度域，每个域包含了共享同一个 L2 高速缓存的所有核心；再向上分别是处理器调度域（包含了同一个处理器中的所有核心）、NUMA 节点调度域（包含了一个 NUMA 节点中所有核心）、全系统调度域（包含了整个系统中所有核心）。内核在启动时会将 CPU 根据其拓扑结构归为不同的域。不同的域有不同的负载均衡周期，越下层的域，任务在核心之间迁移的开销越低，执行负载均衡越频繁。到 NUMA 节点域这一层，负载均衡就很少执行。通过划分调度域的方法，就可以在实现一定的负载均衡的同时，避免在 NUMA 节点之间频繁迁移任务。

12.6 思考题

1. 在目录式缓存一致性协议中，为了避免单一的共享目录成为瓶颈，目录该如何设计？

2. 在现代处理器的缓存一致性协议的设计中，缓存行除了我们介绍的 MSI 状态，通常还添加了其他的状态如独占未修改状态（Exclusive）[9]，该状态表示只有当前核心拥有该缓存行的拷贝。添加新的状态会显著提高处理器设计的复杂性，请分析为何现代处理器会采用这种设计？

3. 在 MCS 锁（代码片段12.5）的`lock`流程中，是否需要在没有人拿锁时（`tail == NULL`，第 16 行），添加`me->flag = GRANTED`以表示当前线程被允许进入临界区？

4. 如果分别采用 TSO 一致性模型与弱序一致性模型，在 MCS 锁（代码片段12.5）的`lock`流程中，`tail->next = me`前（第 16、17 行）是否需要添加内存屏障？为什么？

5. 对于代码段8.8介绍的排号锁，是否也会出现可扩展性问题？如何简要修改该代码来使其可扩展性变好？

6. 在偏向写者的读写锁中，读者需要累加一个全局计数器来避免写者进入临界区。因此这个全局计数器所在缓存行会被不同核上的读者与写者竞争，其暴露的开销会在读者的关键路径上。请描述一下如何修改设计可以避免这个问题？

7. 既然对单一缓存行竞争会产生严重的可扩展性问题，那我们是不是应该将所有的共享数据都放在不同的缓存行来避免这个问题？

8. 我们在12.4.2节介绍的 cohort 锁是否能直接用在操作系统内核中？你认为有哪些方面的问题需要考虑？

9. 开放问题：在本章介绍了多核硬件的各种局限性之后，请举具体例子，什么样的应用适合使用多核多处理器进行计算？什么样的应用更适合使用单核进行计算？

参考文献

[1] Linux memory policy. https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt.

[2] Linux scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-domains.txt.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[4] ARM. Programmer's guide for armv8-a. https://static.docs.arm.com/den0024/a/DEN0024A_v8_architecture_PG.pdf, 2015.

[5] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[6] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: a general technique for designing numa locks. *ACM SIGPLAN Notices*, 47(8):247–256, 2012.

[7] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364, 2010.

[8] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[9] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, 1984.

[10] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193, 2015.

多核与多处理器：扫码反馈