6.7 Cache 相关问题：

高效合理地使用 Cache 对系统性能的影响非常重要。对于串行程序来说，现有的 cache 机制一般都能够正常工作，而对于并行程序来说，cache 的高效合理使用还存在着一些值得关注的问题。

6.7.1 伪共享

大多数高性能处理器在慢速内存和 CPU 高速寄存器之间插入一个高速缓存缓冲区。访问内存位置要求将包含该内存位置的一部分实际内存（缓存代码行）复制到高速缓存。随后可能在高速缓存外即可满足对同一内存位置或其周围位置的引用，直至系统决定有必要保持高速缓存和内存之间的一致性。

然而，同时更新来自不同处理器的相同缓存代码行中的单个元素会使整个缓存代码行无效，即使这些更新在逻辑上是彼此独立的。对缓存代码行的单个元素进行更新会将此代码行标记为无效。其他访问同一代码行中不同元素的处理器将看到该代码行已标记为无效。即使所访问的元素未被修改，也会强制它们从内存或其他位置提取该代码行的较新副本。这是因为基于缓存代码行保持缓存一致性，而不是针对单个元素的。因此，互连通信和开销方面都将所有增长。并且，正在进行缓存代码行更新的时候，禁止访问该代码行中的元素。这种情况称为伪共享。

在执行应用程序时，对占据主导地位的并行循环进行仔细分析即可揭示伪共享造成的性能可伸缩性问题。通常可以通过以下方式减少伪共享：

尽可能多地使用专用数据；
利用编译器的优化功能来消除内存加载和存储。Windows 编译器有一条指令 __declspec(align(n))，可以用于指定按 n 个字节进行对齐。

在特定情况下，当处理较大的问题时由于存在较少共享，可能较难看到伪共享的影响。处理伪共享的方法与特定应用程序紧密相关。在某些情况下，数据分配方式的更改可能减少伪共享。在其他情况下，更改迭代?到线程的映射，赋予每个块的每个线程更多的工作，也可能减少伪共享。

关于伪共享的讨论反映出高效利用单核处理器和高效利用多核处理器之间的密切联系。一般说来，要在单核处理器上高效执行，要进行数据压缩，这样就能够减少访问内存次数。而在多核处理器上，压缩共享数据会导致严重的伪共享问题。解决方法是尽可能多地使用专有数据，在压缩数据以后，为每个线程分配一个私有副本，每个线程访问私有的副本，最后再将每个线程的执行结果合并起来，得到一个总的结果。将这种策略进行推广，就引出了任务窃取策略。当一个线程窃取到一个任务以后，它将可能会引起伪共享问题的共享数据复制一份用作私有副本，然后再计算、合并结果。

6.7.2 存储一致性

对于一个串行程序任一时刻的任一运行副本来说，存储器都有一个良定（well defined）状态，称为顺序一致性。顺序一致性的存储器要满足以下条件：

每个线程的内部操作顺序是确定不变的；
假如所有的 CPU 上的线程都对某一个存储单元执行操作，那么，它们的操作顺序是确定的，即任一线程都可以感知到这些线程同样的操作顺序。

上述条件的含义是指，当多个线程分别在不同的机器上并发执行的时候，只要所有的线程都保持同样的顺序访问存储器，那么，任何有效的交叉访问执行都是可以接受的。在顺序一致性模型中，时间不再是影响一致性的因素，它关心的是：所有线程都必须能够感受到一致的内存访问序列。顺序一致性模型不确保线程的一次读操作可以返回由另一线程所写入的最新值。在没有显式的同步操作的情况下，再一次运行同样的程序不能保证获得同样的结果。

但是并非所有的线程都要求看到所有的写操作的结果，让他们按照顺序看到写操作的结果更是没有必要。这样处理，模型的开销会严重影响应用程序的性能。考虑到运算的中间结果在大多数情况下并没有必须传送出去的必要的具体情况，我们必须对一致性要求作进一步的放松，修改为只有在需要传播写操作的结果的时候才将结果传送出去，除此之外，一切读写操作都完全是并行的。为了达到同步操作的目的，在松弛一致性（relaxed consistency）中引入了同步变量。松弛一致性模型必须满足以下几点条件：

对同步变量的访问满足一致性的要求
对同步变量的访问，只有在以前的写操作在各处都完成之后才能完成。
对数据的操作（读或写），只有在以前的对同步变量的操作完成之才能完成。

第一点说明所有的线程都能以同样的顺序感知到所有对同步变量的访问。当一个线程访问某同步变量时，它会把对该同步变量的访问广播出去，在该线程对该同步变量访问操作成功之前，任何别的线程对同步变量的访问都将被阻塞。第二点说明对同步变量的访问会导致对内存进行刷新的结果。当一个同步访问完成之后，那么，所有先前的写操作可以同时确保完成。当某线程对一个共享数据作了更新之后面，它可以通过同步操作将新值传播出去。第三点说明当一个线程在读一个共享数据（非同步变量）时，通过同步操作，它能获得该共享数据的最新值。

松弛一致性不仅存在于硬件中，编译器也经常会对指令进行重定序。指令重定序对于多数编译器优化来说至关重要。例如，编译器通常会将循环中不变的读操作移动到循环体之外，这样读操作只需要执行一次，而不是在每次迭代中都执行一次。程序设计语言规则一般允许编译器假设代码是单线程的，即使有时候不是这样。对于 Fortran、C 和 C++这类比较老的程序设计语言尤其是这样，在这些语言流行的年代，并行处理器还是一种深奥的东西。对于最新的语言，如 Java 和 C#来说，编译器必须更加谨慎，但是实际上只有出现关键字 volatile 的时候，编译器才会慎重。和硬件重定序不同，编译器重定序会对代码产生影响，即使该代码是在单硬件线程上以时间片轮转方式运行时也是一样。因此程序员必须要监视硬件或编译器进行重定序。

6.7.3 当前 IA-32 体系结构

IA-32 能够实现近似的顺序一致性，因为它是从单核时代发展过来的。关键在于 IA-32 如何支持遗留（legacy）软件。如果严重背离顺序一致性，那么遗留代码就没有用了。但是，一味地强调顺序一致性会导致程序性能很差，所以需要折中考虑，这种折中考虑极大提高了程序性能。可以用两条规则来概括典型的程序设计方法：

松弛以保证更好的性能。一个线程看到另一个线程的读写顺序与原始顺序相同，但对不同单元的读和写来说，读可以超越写。有了这种重定序规则，即使一个线程的读操作位于写主存操作之后，该线程还是能够读取自己的 cache 以完成该读操作。这一规则不涉及“延迟写”的情况。

严格以保证正确性。一条带有 LOCK 前缀的指令就像是一个存储栅栏，任何读写操作都不能越过这个栅栏。这以规则可以避免松弛，维护典型的基于 LOCK 指令的同步过程。此外，指令 xchg 有一个隐式的 LOCK 前缀，可以支持出现 LOCK 前缀概念之前所编写的代码。

这种轻微松弛的存储一致性称为处理器定序（processor order）。为了实现更高的效率，IA-32 还允许某一载入操作在前面的载入操作之前进行，而对程序员来说这是不可见的。但是如果处理器检测到重定序可能会产生某种可见的效应，它就会撤销受影响的指令并重新运行。这样唯一可见的松弛现象就是读操作在其前面的写操作之前执行。

IA-32 的实现在很大程度上都遵循这经典理论，但有趣的是它背离了教科书中称为 Dekker 算法的互斥访问算法。该算法可以在不需要特殊原子操作的情况下实现处理器的互斥访问。图 6.8（a）演示了 Dekker 算法的主要步骤。其中两个变量 X 和 Y 都被初始化为 0。线程 1 写 X 读 Y，线程 2 写 Y 读 X。在一台遵循顺序一致性模型的机器上，无论读和写是否交错，都只有一个线程会读到 0。读到 0 的线程可以访问互斥区域。在 IA-32 以及几乎所有现代处理器上，两个线程都有可能读到 0，因为读可以在写之前完成。此时代码的运行过程图 6.8（b）所示。

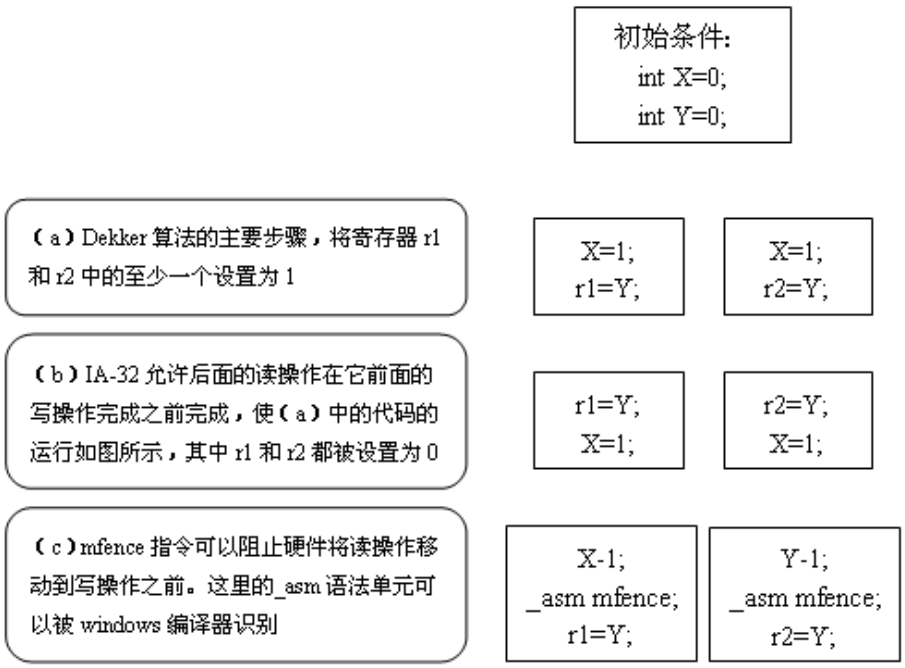

图 6.8 Dekker 算法

图 6.8（c）显示了如何插入一个显式的存储栅栏指令，从而变换代码的运行方式。该栅栏可以阻止读操作在写操作之前执行。下表总结了三种类型的 IA-32 栅栏指令。

表 6.2 三种类型的 IA-32 栅栏指令

助忆符	名称	描述
mfence	存储器栅栏	读写操作都无法越过栅栏
ifence	载入栅栏	读操作无法越过栅栏
sfence	存储栅栏	写操作无法越过栅栏

栅栏可以在需要的时候严格限制访存顺序，以保证正确性。对于延迟写指令，写的顺序可以放松一些，因为其他处理器看到的写指令的完成顺序不需要和这些指令被发出时的顺序相同。有些 IA-32 字符串操作，如 MOVE 和 STOS，可以是延迟写的。较松弛的访存顺序使处理器能够合并写操作，充分提高总线利用效率。但是，需要处理被写入数据的那些处理器可能不希望看到写操作完成的顺序和这些操作被发出的顺序不同，因此生产者应该在通知消费者数据已就绪之前使用一条 sfence 指令。

IA-32 还允许对某个特定的存储器区间变通存储一致性准则。例如，一段允许“写合并”的存储器区间允许处理器在一个缓存区中暂时记录写的内容，之后再以不同的顺序向 cache 或主存提交保存的结果。这样一个存储器区间工作起来就像所有的写操作都是延迟写一样。在实际应用中，为了能够使用遗留代码，多数环境都将 IA-32 系统配置成采用处理器定序模型，因此只有在特殊的环境中才使用 page-by-page 规则。

6.7.4 Itanium 体系结构

Itanium 体系结构不需要支持遗留软件，因此可以采用最新的松弛存储器模型。该模型给于存储系统更高的选择自由度，在理论上可以获得比顺序一致性更高的性能。只要能够正确使用锁来避免数据竞争，一切将在意料之中。但是程序开发人员要编写带有数据竞争的多线程代码，就必须理解存储一致性规则。Itanium 处理器的存储一致性规则虽然比 IA-32 系统更加松弛，但还是很便于记忆的，因为这些规则都很规整。此外，基于 Itanium 系统的编译器对 volatile 的解释方式也遵循大多数传统习惯。

图 6.9（a）给出了一个简单而实际的示例，体现了存储一致性规则。其中两个线程试图通过存储器传递一条消息。线程 1 将消息写入变量 Message，线程 2 读取该消息。同步是通过使用标志 IsReady 实现的。写线程在写入消息之后将 IsReady 置为 1，读线程通过忙等待机制等待 IsReady 被置为 1，然后读取消息。如果写操作或读操作被重定序，那么线程 2 就有可能在线程 1 写消息之前就进行读操作。图 6.9（b）显示了 Itanium 体系结构可能进行的读写重定序。图 6.9（c）所给出的解决方法是将 IsReady 标志设置为 volatile。volatile 写操作被编译成“释放并存储”，而 volatile 读操作被编译成“获取并载入”。存储操作不允许在“释放”之后进行，也不允许在“获取”之前进行，这样就保证了所需要的顺序。

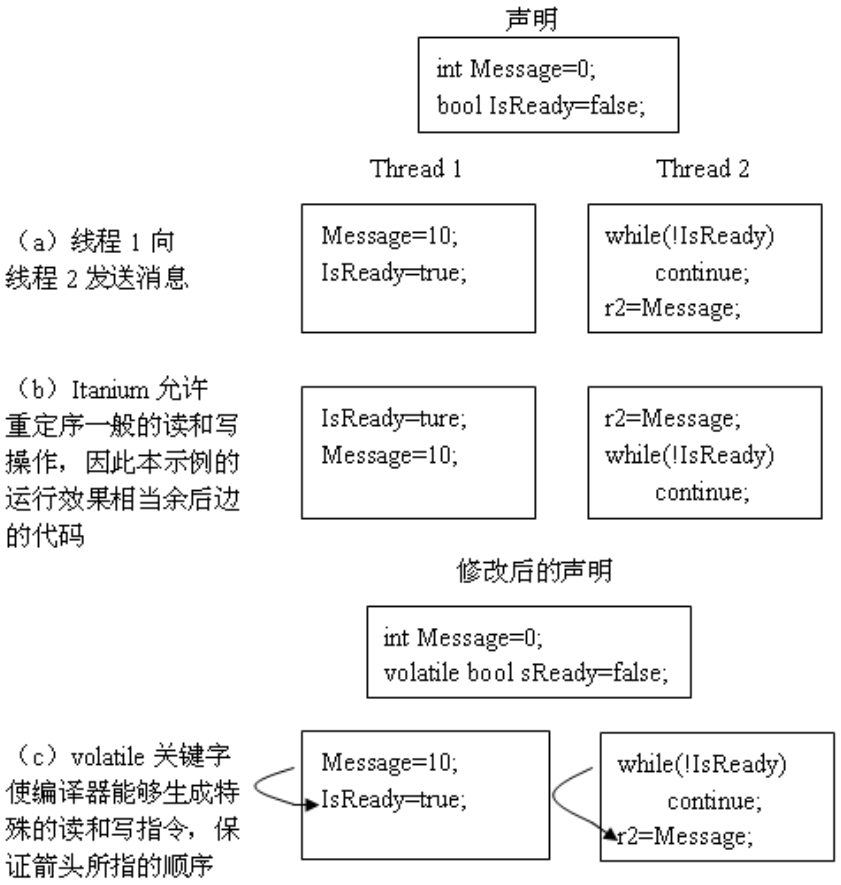

图 6.1 存储一致性规则

Itanium 体系结构的两种松弛存储模型经常被使用。第一种机制是消息传递（message passing）。发送线程写入一些数据，然后想接受线程发信号，通过修改某个标志位告诉接受

线程消息已经就绪。对标志位的修改可能是一次写操作，也可能是其他的某种原子操作。只要发送者在写入数据之后进行释放操作，接收者就可以在读取数据之前进行获取操作，这样就能保证期望的执行顺序。在一般情况下，将标志声明为 volatile 或使用具有所期望的获取/释放特征的原子操作就可以保证这些约束条件不被破坏。

第二种机制是存储笼（memory cage）。一个存储笼起始于一个获取栅栏，终止于一个释放栅栏。这些栅栏将所有的存储操作都限制在笼中。但是，要知道存储笼只能保证将操作限制在其中，不能保证外边的操作不进来。对于不相交的笼来说，指令的重定序有可能使这些笼重叠，因为笼起始处的获取栅栏可能会飘移到前一个笼结束处的释放栅栏之前。同样的原因，想通过使用获取读操作和释放写操作来修正 Dekker 算法也是徒劳的——要修正算法就需要防止读操作飘移到写操作前面，而获取操作总会漂移到释放写操作前面。正确的修正方法是添加一个完全的存储栅栏，例如，调用__memory_barrier()。

还有一个关于栅栏的巧妙示例，那就是被广泛使用的双重检查（double-check）机制。该机制一般用在多线程代码的惰性初始化（lazy initialization）过程中。下图显示了 Itanium 体系结构上双重检查的一种正确实现方式。其中关键的特点是将标志位设置成 volatile，这样编译器就可以插入正确的获取和释放栅栏。双重检查实际上是一种消息传递机制，其中的消息是一种初始化了的数据结构。这种实现机制就 ISO C 和 C++标准来说，不能保证正确性，但对 Itanium 体系结构来说是正确的，因为 Itanium 处理器对 volatile 读操作和写操作的解释隐含了栅栏。

有一种常见的错误分析，那就是认为外层 if 和 read data structure 之间的 acquire 栅栏是多余的，因为似乎硬件一定会在 read data structure 之前执行 if 语句。但是某个普通的读操作实际上有可能会迁移到 if 之前，只要读取的 cache 行和 if 之前另一个读操作所读取的 cache 行相同。同样地，没有了栅栏，一些功能强大的编译器就会将读操作移动到 if 之前，作为一种推断读操作，因此，acquire 栅栏在这里至关重要。

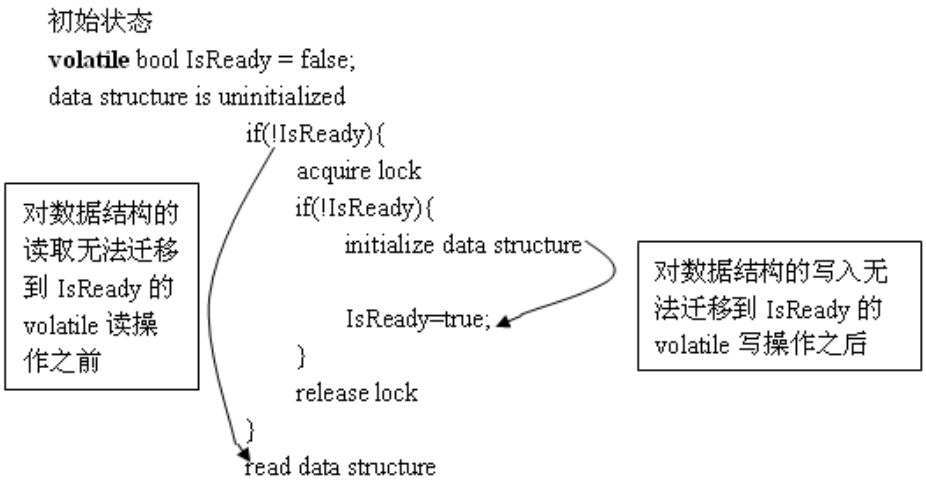

图 6.2 栅栏的双重检查机制

6.7.5 高级语言

在使用高级语言编写可移植代码的时候，处理存储一致性最简单的方法就是使用语言本身的

同步原语，这些原语一般都有正确的内建栅栏。只有在开发人员使用自己的同步原语时，才会出现存储一致性问题。如果必须使用自己的同步机制，那么一致性规则将取决于语言和硬件。下面是一些指导原则：

C 和 C++目前还没有可移植方案。
.NET 使用 volatile 进行声明，就像在 Itanium 体系结构下那样，代码对于任何体系结构来说都是可移植的。
Java 最新的关于 Java 存储管理机制的 JSR-133 修订规范，类似于 Itanium 体系结构上的.NET 环境，要使用 volatile 进行声明。