Memory Model: 从多处理器到高级语言

前言

- 谁需要关心这个主题？
  - 实现同步原语、Lockless算法的并行计算工程师
  - 实现操作系统内核或驱动，和DMA设备打交道的系统工程师
  - 实现高级语言Memory Model的编译器工程师
  - 实现处理器Memory Model的CPU工程师
- 如何阅读本文？

  阅读的时候注意本文的组织结构，第一遍读的时候跳过斜体的技术细节
- 除基本介绍外，本文还会回答以下问题：
  - x86为什么允许Store-Load乱序，而不是其他？*见TSO的优点*
  - std::memory_order_seq_cst非常慢、会严重损害性能？*见WO*
  - std::memory_order_acquire和std::memory_order_consume的区别？后者换作std::memory_order_relaxed会怎样？*见Dependent Loads*
  - std::memory_order_acq_rel和std::memory_order_seq_cst的区别？*见Store Atomicity和IRIW*

基础知识

- **共享存储器多处理器(Shared-Memory Multiprocessor)**

  SMP是通过一个共享的地址空间进行通信和协调的多处理器系统，与之相对的概念是`集群(Cluster)`，后者每个节点都有独立的地址空间。SMP又包括：
  - `NUMA(Non-Uniform Memory Access)`：大量的处理器通过互连网络连接在一起，每个节点访问本地存储器时延迟较短，访问其他节点有不同的延迟
  - `UMA(Uniform Memory Access)`：又叫`SMP(Symmetric Multiprocessor)`，所有的处理器通过总线连接在一起，访问一个共享的存储器，每个节点的存储器访问延迟相同。目前常见的桌面多核处理器就是这种结构
- 现代处理器中的技术[2]
  - `流水线(Pipeline)`：每个指令的执行需要多个步骤，线代CPU通过流水线的方式允许同时执行多个指令，从而提高功能单元的利用率和系统总吞吐。支持流水线的CPU的`IPC(Instructions Per Second)`可以达到1，哪怕每条指令实际上需要多个时钟周期才能完成
  - `动态分支预测(Dynamic Branch Prediction)`：带流水线的CPU需要每个时钟发射1条指令，但只有分支指令执行结束后才能确定下条指令是什么，这就导致`流水线停顿(Pipeline Stall)`。为避免分支指令导致的流水线停顿，一种对策是分支预测，即在发射分支指令之后，马上预测下条指令的地址并发射，如果分支指令执行结束后发现预测错误，则撤销之前的操作取正确的指令重新发射。这里预测失败导致的撤销开销，叫`分支预测惩罚(Mispredict Penalty)`，由于现代系统的分支预测正确率很高，摊还后的惩罚开销往往可以接受。动态分支预测是基于分支指令历史进行的，现代CPU的预测正确率在大部分场合

可以高达95%以上；相对的，静态分支预测是基于固定分支选择策略、源码中的Hint，或根据编译器的得到的Profile信息来完成的

- `动态多发射(超标量，Superscalar)`：为更好的利用富裕的功能单元，CPU希望IPC能够超过1，这就要求每个时钟发射多条指令。支持超标量的处理器，需要处理同时发射的多条指令间的数据依赖关系，这个复杂性限制了动态发射窗口的大小。与之相对的是静态多发射，即由编译器或程序员往一个`发射包(Issue Packet)`中填充多条无关指令，然后同时发射和执行，典型的例子是`超长指令字(Very Long Instruction Word)`体系结构
- `乱序执行(Out-of-Order Execution)`：当前面的指令由于特种类型的功能单元不足、存储器延迟或操作数没有计算出来，必须停顿时，CPU可以发射后续的无关指令，从而乱序执行。乱序指令有效的提高了CPU利用率，掩盖了各种停顿
  - `寄存器重命名(Register Renaming)`：CPU通过寄存器重命名的方式使得物理寄存器数目超过指令集中的逻辑寄存器，缓解了如x86指令集中的寄存器不足的问题。*寄存器重命名另一大作用是，避免由于`假数据依赖(False Data Dependence)`导致的停顿；具体来说，寄存器重命名解决了`WAW Hazard`和`WAR Hazard`*
- `推断执行(Speculative Execution)`：支持动态分支预测和乱序执行的处理器，需要保留一个`重排序缓冲区(Reorder Buffer)`，用来对乱序执行的指令进行`顺序提交(In-Order Commit)`。重排序缓冲区为推断失败时的撤销提供了解决方案，只需要清空分支指令后的所有指令即可。另外，顺序提交也为`精确异常(Precise Exception)`提供了基础，这是操作系统中断和`缺页故障(Page Fault)`处理的关键。推断执行的指导思想是“加速大概率事件”
- `写缓冲区(Writer Buffer)`：CPU在写存储器时，不直接访问存储器或Cache，而是将要写的数据放入一个写缓冲区，然后继续执行后面的指令，这缓解了写存储器导致的停顿
- `硬件多线程(Hardware Multithreading)`：上面列举的优化策略都旨在改进`指令级并行(Instruction-Level Parallelism)`，另一种提高CPU利用率、掩盖停顿的做法是硬件多线程，即为每个CPU准备两套寄存器和指令计数器，同时从两个逻辑线程取指令并轮询发射，由于这两个线程的指令间没有依赖，并不会引入额外的停顿。支持推断执行的硬件多线程技术又叫`同时多线程(Simultaneously Multithreading)`

Cache Coherence[2,4]

多处理器系统中(这里讨论的处理器，也包括DMA设备)，为减少存储器访问延迟，会为每个处理器添加本地的Cache(比如Core i7-6700K每个Core各有32KB的`Instruction Cache`和32KB的`Data Cache`)，引入本地Cache会导致数据多副本问题：某处理器更新了存储器中的一个块，同一个块可能在其他处理器的Cache中还有过期副本。多处理器需要专门的硬件来实现Cache Coherence Protocol。*本文仅仅讨论`Write-Back` & `Write-Invalide` & `Snooping-Based`的Cache，对`Write-Through`/`Write-Broadcast`/`Directory-Based`的Cache的讨论类似*

- `Cache Coherence Protocol`
  - `MSI协议`：每个Cache Line上有几个标记位用来标志其处于Modified/Shared/Invalid中的某个状态，当处理器读写该Cache Line时，会根据其状态进行状态迁移并发送相应的协议消息以保持多副本数据的一致性
    - *`Shared状态`：处理器读操作引起的Read Miss会令该Cache Line以Shared状态从存储器读入本地Cache中；如果之前该Cache Line以Modified状态被某处理器持有，那监听到这个Read Miss的处理器用它持有的该Cache Line最新的副本响应源处理器，并更新存储器*
    - *`Modified状态`：处理器写操作引起的Write Miss会令Cache Line以Modified状态从存储器读入本地Cache中；如果之前该Cache Line被一个或多个处理器以Shared状态持有，则写操作处理器将向他们发送`Invalidate Message`令它们持有的Cache Line失效；如果之前该Cache Line被某处理器以Modified状态持有，则写操作处理器向它发送`Read-Invalidate Message`，目标*

处理器收到消息后将其持有的Cache Line标记为Invalid并回应以最新的数据副本同时更新存储器

- `Invalid状态`：*收到Invalidate或Read-Invalidate Message的处理器会将对应的Cache Line置为Invalid状态*

- `MESI协议`：在MSI协议的基础上，从Shared状态中分离出`Exclusive状态`来避免独占Cache Line的处理器第一次写Cache Line时发出的Coherence Message，从而减少总线流量*(即，相比Shared到Modified状态的迁移，第一次写独占Cache Line时进行的是Exclusive到Modified的迁移，不必发送Invalidate Message了)*

- `写缓冲区（Write Buffer）`：写缓冲区不仅可以在单处理器中被用来掩盖写延迟，也可以用来掩盖Cache Coherence Protocol的延迟。*处理器为确保其他处理器看见自己的写操作，需要等待其他处理器在处理完自己发出的Invalidate Message后回应以Acknowledge Message，这个时间可能是几十上百个时钟周期，而利用Write Buffer，处理器可以在将一个写操作放入Write Buffer的同时就发出Invalidate Message，在收到Acknowledge Message后才从Write Buffer中移除并以Modified状态写入自己的本地Cache，而在收到来自其他处理器的回应前可以继续执行其他指令*

- `失效队列（Invalidate Queue）`：当处理器工作负载很高时，可能来不及处理、回应来自其他处理器的Coherence Message，从而将等待回应的其他处理器阻塞，而利用一个消息缓冲区缓存这些请求并立刻回应源处理器表示消息已收到，可以提高响应速度。*处理器在收到Invalidate Message后可以先将其放入Invalidate Queue并立刻回应以Acknowledge Message，并在之后不忙或发生Cache Miss的时候再回来处理Invalidate Queue中缓存的请求*

编译器中的优化技术

编译器被允许在不改变单线程程序执行结果的前提下，进行各种优化

- `公共子表达式删除(Common Subexpression Elimination)`[17]

```
a = b * c + g;   //---------->    tmp = b * c;
d = b * c * e;   //  rewrite      a = tmp + g;
                 //               d = tmp * e;
```

- `死代码删除(Dead Code Elimination)`

```
while (!flag);  //----------->    loop:
                //                jmp loop
```

- `寄存器分配(Register Allocation)`：下例将对g的2次读取优化成了1次

```
a = g;          //----------->    load r1, mem1
b += a;         //  rewrite       add r2, r2, r1
a = g;          //
c += a;         //                add r3, r3, r1
```

- `指令调度(Instruction Scheduling)`：下例重排的后一条指令不必等待前一条的结果减少了停顿*(这里的静态流水线调度缓解了`RAW Hazard`)*

```
load r0, mem0   //                load r0, mem0
mul r1, r1, r0  //------------>    load r2, mem2
store mem1, r1  //  rewrite       mul r1, r1, r0
load r2, mem2   //                mul r3, r3, r2
mul r3, r3, r2  //                store mem1, r1
store mem3, r3  //                store mem3, r3
```

Memory Model

- 什么是**Memory Model**

  `Memory Model(Memory Consistency/Memory Consistency Model)`是系统和程序员之间的规范，它规定了在一个共享存储器的多线程程序中的存储器访问应该表现出怎样的行为。这个规范影响了系统的性能，因为它决定了多处理器/编译器能应用哪些优化；也影响了`可编程性(Programmability)`，因为多线程程序的正确性取决于Memory Model，从而约束了程序员的编程方式

  - 一些解释：
    - 运行在单处理器(且没有DMA设备等)上的单线程程序无需考虑Memory Model问题，因为处理器/编译器的优化都保证对程序员透明，即程序看起来像是按自然顺序(Program Order)执行的
    - 一般讨论Memory Model中的读写乱序，是指对任意地址读写的乱序；但从实现层面讲，由于系统必须保证乱序对执行读写的线程透明，而相同地址的读写存在`数据相关Data Dependence`，所以系统实际上只能对不同地址的读写进行乱序
    - Memory Model的讨论，一般分为两层：汇编语言中的Memory Model由具体多处理器规定和实现，不可移植；高级语言中的Memory Model由高级语言标准来规定，由编译器和目标多处理器共同实现，可移植

- 为什么需要**Memory Model**

  在单线程程序中，编译器的各种优化如冗余代码消除、`循环融合(Loop Fusion)`、指令调度等技术，会重写代码造成存储器访问顺序变化，而寄存器分配会改变存储器访问次数；类似的，处理器的乱序执行机制也会通过让后一条指令先执行，来掩盖前一条指令导致的流水线停顿和存储器停顿。尽管会改变不同地址存储器的访问顺序，但系统的这些优化对程序员是透明的，看起来程序仍然是在顺序执行

  然而在多线程程序中，编译器和多处理器并无手段自动发现多个线程间的协作关系，使得那些可能改变存储器访问顺序和次数的优化，同时对多个线程透明。没有程序员的帮助，要保持多线程程序的正确性，系统只能禁用这些作用在共享存储器上的优化，而这将严重损害性能

  为最大限度保留编译器和多处理器的优化能力，同时使多线程程序的执行结果是可预测的，系统需要程序员的帮助。最后的方案是，系统提供所谓Memory Model的规范，程序员通过规范中同步设施(各种`内存屏障(Memory Barrier)`和Atomic指令)来标记多个线程间的协作关系，使得不仅是单线程，系统的优化对多线程程序也将透明

  `互斥锁(Exclusive Lock)`是被最广泛支持的同步机制，编译器和多处理器会确保基于锁同步的多线程程序看起来就像是有多个同时执行的顺序线程。而一旦离开锁的庇护，程序员要么直面各种优化作用下的混乱世界，要么和实现`同步原语(Synchronization Primitives)`的系统工程师站在同一起跑线，捡起Memory Model这个更细粒度、更微妙的武器，在乱序优化的多线程世界中重建秩序

- 几个反直觉的例子

  当没有正确进行线程间同步时，以下例子都是可能的，它们反映了某些Relaxed Memory Model的行为。

  - Store-Store Reorder 或 Load-Load Reorder

```
{ data == 0, flag == 0 }
//  thread 0                thread 1
//---------------------------------------
    store data, 1           loop:
    store flag, 1           load r0, flag
                            beq r0, 0, loop
                            load r1, data
```

可能r1 == 0。本例在ARM上能重现

- Store-Load Reorder

```
{ x == 0, y == 0 }
//  thread 0                thread 1
//---------------------------------------
    store x, 1              store y, 1
    load r0, y              load r1, x
```

可能r0 == 0 && r1 == 0。本例在ARM/x86上能重现

- `Dependent Loads` Reorder[5]

```
{ A == 1, B == 2, C == 3, P == &A, Q == &C }
//  thread 0                thread 1
//---------------------------------------
    B = 4;
    BARRIER;
    P = &B;
                            Q = P;
                            D = *Q;
```

可能Q == &B && D == 2。本例在DEC Alpha上能重现

- `Non-Causality` / `Non-Transitivity`[3]

```
{ flag0 == 0, flag1 == 0 }
//  thread 0          thread 1          thread 2
//-------------------------------------------------------
    store flag0, 1
                      loop:
                      load r0, flag0
                      beq r0, 0, loop
                      BARRIER
                      store flag1, 1
                                        loop:
                                        load r1, flag1
                                        beq r1, 0, loop
                                        BARRIER
                                        load r2, flag0
```

可能r2 == 0。本例在不支持Causality的系统中能重现

- `IRIW(Independent Read Independent Write)`[3]

```
//  thread 0            thread 1            thread 2            thread 3
//---------------------------------------------------------------------------
    store data1, 1      store data2, 1
                                            load r1, data1      load r3, data2
                                            BARRIER             BARRIER
                                            load r2, data2      load r4, data1
```

在r1 == 1 && r3 == 1的前提下，可能r2 == 0 && r4 == 0，即thread 2和3看见了不同的写顺序。本例在不支持`Atomic Store`的系统中能重现，比如某些NUMA和带SMT的UMA系统

- **Memory Model由哪些属性构成**

  Memory Model主要规定不同地址上的读写操作在其他处理器看来会否乱序，以及，一个写操作是否同时被其他处理器观察到

  - Memory Ordering
    - `Load-Load` Order：不同地址上的读操作会否乱序
    - `Load-Store` Order：读操作和后面另一个地址上的写操作会否乱序
    - `Store-Load` Order：写操作和后面的读操作会否乱序
    - `Store-Store` Order：不同地址上的写操作会否乱序
    - `Dependent Loads` Order：当第二条读操作的地址取决于前一条读操作的结果时，会否乱序
  - `写原子性(Store Atomicity)`

    写原子性是指，处理器的写操作是否同时被所有处理器看到。根据写操作的同时性，从弱到强排序：

    1. Load Other's Store Early && Non-Causality：允许写操作被自己及个别其他处理器先看到，不支持Causality。写序列可能以不同顺序被多个处理器观察到
    2. Load Other's Store Early && Causality：允许写操作被自己及个别其他处理器先看到，支持Causality
    3. Load Own Store Early：只允许写操作被自己先看到。写序列以相同顺序被多个处理器观察到
    4. Atomic Store：所有处理器同时看到写操作

- **多处理器怎样影响Memory Model的属性**

  现代处理器可能出于优化的考虑，放松下面这些限制的一条或多条：

  - Memory Ordering[4]
    - Load-Load Order：当前一条Load指令因为操作数未就绪*(RAW Hazard)*或Cache Miss而必须等待时，乱序执行的CPU可能先执行后一条Load指令以掩盖停顿。*如果Memory Model不允许这种Reorder或程序员在两次Load之间插入了一条Barrier指令，那第一条Load执行完毕后，如果发现有Invalidate Message令后面要Load的Cache Line失效，就应该通过清空ROB撤销乱序执行的Load和后续指令然后重新执行*
    - Load-Store Order：同上，当前一条Load因为各种原因等待时，后一条Store指令可能被先执行。*如果Memory Model或Barrier指令限制这种Reorder，在推断执行的CPU中往往不用做任何事，因为Store指令是在提交阶段才更新存储器*
    - Store-Load Order：当CPU和Cache间有写缓冲(Write Buffer)时，写操作会被放入写缓冲而不是更新Cache，这可能导致Load指令执行过后，写操作仍然对其他处理器不可见。*如果Memory Model或Barrier指令限制这种Reorder，这要求在执行读操作之前，先Flush整个写缓冲，令结果写到Cache中，这里的开销非常大。由于开销非常大，现代处理器都允许这种Reorder，且只有Full Barrier限制这种Reorder*

        - Store-Store Order：当CPU和Cache间有一个Non-FIFI或Coalescing的写缓冲时，如果前一个写操作导致Cache Miss或和其他写操作合并(因为写同一条Cache Line)，后一条结果可能先写入Cache。*如果Memory Model或Barrier指令限制这种Reorder，则要求采用FIFO的写缓冲，或在遇到Barrier指令时Flush整个写缓冲*
        - Dependent Loads Order[4,5]：即使处理器在语句"obj->field = new_value"和"g_obj = obj;"间插入Barrier指令，强制Flush写缓冲区，另一个处理器的相关读操作"obj = g_obj; value = obj->field"在取得新对象指针的同时仍然可能看到旧的字段值，因为新的字段值还在处理器的Invalidate Queue里面，要稍后才能处理。*更详细的描述下这种Corner Case：假设最初obj->field对应的Cache Line被CPU0和CPU1以Shared状态持有，g_obj对应的Cache Line以Exclusive状态被CPU0持有；CPU0写obj->field并用Barrier指令Flush写缓冲，会导致CPU1将一条Invalidate Message放入Invalidate Queue中并仍然以Shared状态持有obj->field，而CPU0以Modified状态持有更新过的obj->field；然后CPU0再更新g_obj，其对应的Cache Line会变成Modified状态；接下来，CPU1读取g_obj的Cache Miss被CPU0监听到并回以最新的g_obj值，但CPU1再尝试读取obj->field会取得旧的值，因为它的Invalidate Queue还有缓存的消息没被处理。在CPU1执行的两条Load之间插入一条Barrier指令可以强制CPU1处理Invalidate Queue中的所有消息，从而避免读取到旧的值。DEC Alpha是历史上唯一会遭遇这种问题的处理器，这个Corner Case影响了Linux Kernel的Data Dependency Barrier和C++标准中的std::memory_order_consume的设计；现代处理器不再有这种问题，所以对应的Barrier会生成空操作(但仍可能影响编译器优化[16])，至于这些处理器怎么避免Invalidate Queue导致的滞后，本文并未考证出结果(一种猜测：在每次最外层本地Cache发生Miss时处理Invalidate Queue中的所有消息?)*
    - 写原子性(Store Atomicity)[1,3]
        1. Load Other's Store Early && Non-Causality：在NUMA和支持硬件多线程的UMA中，Cache Coherence Message可能先抵达较近的逻辑处理器，从而导致写操作被部分处理器先看到。*对照前面的例子，导致Non-Causality的是，thread 0对flag0的写操作比thread 1对flag1的写操作更晚被thread 2观察到，Invalidate Queue是一种可能的原因：假设最初flag0以Shared状态被所有CPU持有，flag1以Exclusive状态被CPU 1持有；CPU 0对flag0的写操作会导致CPU 1和CPU 2各收到一条Invalidate Message，然后CPU 1处理了该Message观察到了flag0的新值并将自己持有的flag1对应的Cache Line切换到Modified状态，而CPU 2因为Cache Miss从CPU 1处取得flag1的新值时，flag0对应的Invalidate Message可能还没被他处理，从而读到flag0在自己本地Cache中的旧值，这就导致因果性/传递性的丧失*
        2. Load Other's Store Early && Causality：一般通过类似Acquire/Release的Barrier指令可以获得Causality。*比如，就上一段的例子，thread 2在读取到flag1的新值后，执行一条Barrier指令来处理所有Invalidate Message，从而确保观察到flag0的当前值*
        3. Load Own Store Early：这里要求除自己外的其他处理器同时看到写操作，开销很大，这就是不严格的所谓Atomic Store，鉴别这种情形的就是上面的例子IRIW。*如果Memory Model或Barrier指令强制这种原子性，这就要求，当所有处理器对写操作发出的Invalidate Message回应以Acknowledge Message后，当前处理器才能把写地址所在的Cache Line切换到Modified状态写入本地Cache，这个延迟可能是几十上百周期，写缓冲的重叠执行和Invalidate Queue引入的延迟处理能缓解这个延迟*
        4. Atomic Store：该情形甚至不允许转发自己的写缓冲中的值，很少见，所以更多是理论价值

- **编译器怎样影响Memory Model的属性**

  在不改变单线程执行结果的前提下，编译器倾向于进行各种代码变换来实施优化，这些优化会改变存储器访问的顺序和次数，故前面列举的Load-Load/Store-Load Reorder等都可能发生。另外，一般来说，相同地址的内存访问受Data Dependence约束往往无法重排，但是一些激进优化也可能导致Dependent Loads Reorder[16]

- **Memory Consistency和Cache Coherence的区别**

Cache Coherence是多处理器的本地Cache导致多个数据副本在Cache和存储器间不一致的问题，Memory Model是多处理器和编译器优化导致存储器操作被多个处理器观察到的顺序不一致的问题。

此外，Memory Consistency和Cache Coherernce还有这些明显区别[3]

- 前者的研究对象是多个地址，后者的研究对象是单个Cache Line
- 就正确性而言，前者对程序员可见，后者对程序员透明，尽管后者影响性能
- Memory Model可以实现在只有Incoherent Cache甚至没有Cache的多处理器系统上

- **理想的Memory Model**
  - 多处理器和编译器对Memory Model的诉求

    对多处理器而言，允许所有存储器操作的乱序可以提供最多的优化机会，哪怕其中一些顺序它能高效实现；允许部分处理器先观察到写结果，能够减少在Cache Coherence Protocol上阻塞的时间。类似的，编译器优化也期望能够自由地对代码做变换，只要这种变换不改变单线程执行的结果。故，多处理器和编译器都倾向于提供一种允许任意乱序、没有原子写要求的Memory Model
  - 程序员对Memory Model的诉求

    程序员需要的Memory Model是，运行在多处理器系统上的共享内存多线程程序，看起来就像是并行或交错的多个顺序执行的线程，没有任何乱序、写操作也立即全局可见，同时每个线程和单线程程序运行得一样块，多线程的整个程序性能随处理器个数伸缩
  - 调和二者的矛盾

    理想的Memory Model，能够尽量满足上面系统和程序员的诉求：允许各种乱序、非原子写，但只要程序员按一定的模式来组织程序、协调多个线程的通信，那这些混乱都是透明的。下面介绍一种Memory Model，它对系统和程序员都足够友好。
    - `SC for DRF programs`[3]

      当两个线程同时访问一个地址，其中至少一个是写操作时，我们说发生了`Data-Race`。有一类良好组织`Data-Race Free`程序，运行在`Sequential Consistency`和`Relaxed Consistency`的Memory Model上都能得到一样的结果，这类程序的行为可以仅仅通过程序顺序(Programm Order)去推断，我们说这类程序叫`SC for DRF programs(Sequential Consistency for Data-Race Free programs)`。*也就是说，SC for DRF的程序能用Sequential Consistency的Memory Model去推断正确性，却同时具备Relaxed Memory Model的性能*
    - 基于存储器分类的Memory Model
      - 存储器分类
        - `私有数据(Private Data)`：简称p_data，只被固定线程访问的数据。没有Data-Race、不需要Cache Coherence；其上的操作可以进行任何乱序，包括跨越s_data和sync_var的读写。*p_data对应的Cache Line一般常驻在特定处理器的本地Cache中，除非操作系统为负载均衡执行Migration。编译器能够识别局部变量但却无法很好识别thread local的堆数据；部分处理器如ARM允许将存储器标记为非共享的*[7]*，从而禁用Cache Coherence并执行各种优化*
        - `共享数据(Shared Data)`：简称s_data，允许多个线程访问，但任意时刻只被一个Owner持有。没有Data-Race，需要Cache Coherence；其上的读写可以进行任何乱序，但不能跨过sync_var。*只读的s_data会存在于多个处理器的本地Cache中，读写的s_data会随着Onwer的变化在多个处理器的本地Cache中迁移。在多处理器和高级语言中默认的存储器就是s_data*
        - `同步变量(Synchronization Variable)`：简称sync_var，允许多个线程同时访问。有Data-Race，需要Cache Coherence；sync_var的读写之间不能进行乱序。*sync_var会被同时用于读写，在`高争用(High Contention)`的情况下对应的Cache Line会频繁出现在多*

个本地Cache中，然后因为特定处理器的写操作而失效。编译器通过高级语言提供的atomic或volatile声明识别出sync_var，进而生成Barrier指令来限制sync_var之间或sync_var与s_data之间的乱序，ARM也提供了Strongly-Ordered的声明以禁用sync_var之间的乱序

利用上面的存储器分类，我们很容易实现锁操作，从而得到程序片段如"access p_data; lock(&sync_var); access s_data; unlock(&sync_var); access p_data "，可以看到，中间`临界区(Critical Section)`中对s_data的访问可以乱序，但不能跨过lock/unlock，两边对p_data的访问可以任意乱序包括调度进临界区。稍后我们会看到，这里给的Memory Model类似于WO

多处理器中的 **Memory Model**

- **强一致性模型(Strong Consistency Model)**
  - `SC(Sequential Consistency)`
    - 定义：将一个多线程程序各线程的操作按程序顺序(Program Order)交错在一起得到一个程序，如果这个单线程程序的执行结果和原来的多线程程序一样，我们就说执行程序的这个多处理器系统的Memory Model是Sequential Consistency的
      - *Leslie B. Lamport定义的SC[18]：He first called a single processor (core) sequential if "the result of an execution is the same as if the operations had been executed in the order specified by the program." He then called a multiprocessor sequentially consistent if "the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program."*
    - Memory Model的属性
      - LL/LS/SL/SS/DL Reorder：不允许
      - Store Atomicity：Load Own Store Early
- **弱一致性模型(Relaxed Consistency Model / Weak Ordering)**

  在SC的基础上，我们逐步放松各个属性的限制，会依次得到各种弱一致性模型

  首先允许SL Reorder：
  - `TSO(Total Store Order)`
    - Memory Model的属性
      - SL Reorder：允许
      - LL/LS/SS/DL Reorder：不允许
      - Store Atomicity：Load Own Store Eearly
    - 优点
      - 推断执行的CPU能高效实现TSO
        - Load-Load Order：保持。*前一个Load遇到Cache Miss的情况下，允许后一个Load先执行，但如果前一个Load返回发现Invalidate Queue当中的消息将使得后续的Load失效，则清空ROB撤销乱序执行重新发射指令。*
        - Load-Store Order：保持。*推断执行CPU是顺序提交的，而Store指令的写存储器发生在提交阶段，此时Load指令已经Retire*
        - Store-Load Order：乱序。*要使得Store指令先更新存储器，必须禁用写缓冲区或者在Load指令之前Flush写缓冲区*

        - Store-Store Order：保持。*利用FIFO、Non-Coalescing的写缓冲区，每条Store指令是顺序更新存储器的*
    - Load/Store分别等效于Acquire/Release操作，实现锁时无需Barrier

      Acquire指的是一个读操作，其后的Load/Store不允许Reorder到前面，而Release是一个写操作，其前的Load/Store不允许Reorder到后面。因为在TSO中，LL、LS乱序是不允许的，故Load指令直接可以用作Acquire；类似的，因为LS、SS乱序是不允许的，Store指令也可被用作Release。

      实现SpinLock等互斥锁时，Acquire操作可以被用于Lock，Release操作可以被用于Unlock，它们一起避免了临界区内的共享存储器的读写被Reorder到临界区外，从而避免了Data-Race。由于在TSO中Load、Store分别等效于Acquire、Release，故在x86等系统中实现锁是不需要Barrier指令的。
  - 评价：这是一种相对较强的Memory Model，被x86采用[21]，程序员可以简单地把它抽象成带写缓冲区(Write Buffer)的多处理器系统
- `PC(Processor Consistency)`
  - 和TSO的区别
    - Store Atomicity：Load Other's Write Early
  - 优点：相比TSO，放松了写一致性的要求，降低了Coherence Protocol上的开销

进一步地，允许SS Reorder：

- `PST(Partial Store Order)`
  - 和TSO的区别
    - SS Reorder：允许
  - 优点：相比TSO，允许Non-FIFO和Coalescing的写缓冲区，允许多个Store指令合并或重叠的执行

再进一步地，允许LL/LS Reorder：

- `WO(Weak Ordering)`
  - 定义：在WO中，共享存储器被划分为Data(对应前文s_data)和Synchronizing Variable(对应前文sync_var)。对Synchronizing Variable的读写要保持相对顺序，对Data的读写允许乱序，但不能跨过对Synchronizing Variable的操作。
    - *Michel Dubois等人定义的WO[19]：In a multiprocessor system, storage acceses are weakly ordered if 1) accesses to global synchronizing variables are strongly ordered and if 2) no access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed and if 3) no access to global data is issued by a processor before a previous access to a synchronizing varible has been performed*
  - Memory Model的属性
    - LL/LS/SL/SS Reorder：允许(Data读写)，不允许(Synchronizing Variable读写)
    - DL Reorder：不允许
    - Store Atomicity：Load Own Store Early
  - 评价：WO是一种粗粒度的Memory Model，程序员可以先把所有存在Data-Race的存储器标记为Synchronizing Variable来确保程序正确性，之后再优化，故易于编程。因为一般程序中Data的访问远多于Synchronizing Variable，其上的操作都可乱序，所以WO的性能也相当不错。**WO是一种在性能和可编程性上有很好折中的Memory Model**，本文后面还会多次提起它(在高级语言讨论中)

- `RC(Release Consistency)`
  - 定义：在RC中，共享存储器操作被划分为`Ordinary`(对应前文s_data)和`Special`，其中Special进步一划分为`Sync`(对应前文sync_var)和`NSync`，Sync又包括Acquire和Release。Ordinary操作的乱序不允许向前跨过Acquire、不允许向后跨过Release，根据Special操作之间是Sequential Consistency或Processor Consistency，RC又可以进一步细分为`RCsc`和`RCpc`：(注意后者不要求Store Atomicity)
  - RCsc
    - Memory Model的属性
      - LL/LS/SL/SS Reorder：允许(Ordinary操作)
      - DL Reorder：不允许
      - Store Atomicity：Load Own Store Early
  - RCpc
    - Memory Model的属性
      - LL/LS/SL/SS Reorder：允许(Ordinary操作)
      - DL Reorder：不允许
      - Store Atomicity：Load Other's Store Early && Causality
    - *Kourosh Gharachorloo等人定义的RCpc*[20]*：(A) before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous acquire access must be performed, and (B) before a release access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and (C) special accesses are processor consistent with respect to one another*
  - 评价：当Ordinary操作(对应前文s_data)仅出现在临界区内时，用WO实现的锁和RCsc等价，比RCpc版本多了写原子性；在更复杂的Lockless算法中，RCpc提供了比WO更细粒度的重排序控制。**RC是一种提供细粒度控制的常见Memory Model**，我们之后还会看见它(在高级语言讨论中)
- ARMv7[7]
  - 定义：ARMv7的存储器被划分为`Strongly-Ordered`(对应前文sync_var)、`Device`和`Normal`，其中Normal又分为`Shareable`(对应前文s_data)和`Non-Shareable`(对应前文p_data)。Strongly-Ordered和Device上的操作之间不允许乱序，Normal的操作可以任意乱序。要限制Normal操作的顺序需要显示的Barrier指令
  - Memory Model的属性
    - LL/LS/SL/SS Reorder：允许(Normal操作)，不允许(Strongly-Ordered/Device操作)
    - DL Reorder：不允许
    - Store Atomicity：Load Own Store Early

最后，允许DL Reorder：

- DEC Alpha
  - Memory Model的属性
    - LL/LS/SL/SS/DL Reorder：允许
    - Store Atomicity：Load Own Store Early
  - 评价：DEC Alpha的生命期实际上在2001年已经宣告结束，我们之所以讨论它，是因为它是唯一一款允许Dependent Loads Reorder的多处理器，这使得它有几乎最Weak的Memory Model，而这又促成了Linux Kernel的Barrier设计(Data Dependency Barrier)以及C++标准的memory model设计(std::memory_order_consume)

        - *引自Paul E. McKenney[4]：Alpha is interesting because, with the weakest memory ordering model, it reorders memory operations the most aggressively. It therefore has defined the Linux-kernel memory-ordering primitives, which must work on all CPUs, including Alpha. Understanding Alpha is therefore surprisingly important to the Linux kernel hacker*

高级语言中的Memory Model

Linux Kernel和高级语言标准都定义了自己的Memory Model，其中有专门的同步操作用于线程间协作。编译器负责将这个抽象的Memory Model映射到目标多处理器上，如果多处理器自己的Memory Model相对更强，那么上层Memory Model的同步操作可能退化成普通的存储器访问；如果目标多处理器的Memory Model相对更弱，则部分上层同步操作可能生成处理器提供的Barrier或Atomic指令来强制顺序和写原子性

- **Linux Kernel[5]**

  Linux Kernel的代码早于Memory Model的概念被引进C语言(2011)，所以必须自己面对多处理器下存储器操作乱序的问题。为更好的可移植性，Kernel假设多处理器有一个最弱的Memory Model(不强于DEC Alpha)、编译器也会进行任意乱序。为在这种环境下正确的编写多线程程序和设备驱动，Linux使用了如下Barrier：

  - 编译器Barrier
    - `READ_ONCE(x, value)`和`WRITE_ONCE(x)`：用于存储器的读写，避免这些读写彼此被编译器乱序或优化掉。
  - 处理器Barrier
    - Store Barrier：前后的Store指令不能越过Barrier乱序，搭配Load Barrier或Data Dependency Barrier
    - Load Barrier：前后的Load指令不能越过Barrier乱序，搭配Store Barrier
    - Data Dependency Barrier：依赖于前一条Load指令结果的第二条Load指令不能越过前一条Load乱序，搭配Store Barrier。这是Load Barrier的优化版本，用于仅需要避免读相关乱序的场合，例如Producer更新对象字段后将对象指针传给Consumer使用
    - General Barrier：相当于Store+Load Barrier，搭配Load/Store/Data Dependency Barrier
    - Acquire操作：之后的读写不能向前乱序，搭配Release操作
    - Release操作：之前的读写不能向后乱序，搭配Acquire操作

  其中编译器Barrier是通过C语言的volatile实现的(比如将目标地址转化为用volatile修饰的指针后再读写)，而处理器Barrier根据目标多处理器的Memory Model生成读写指令和必要的Barrier指令，保证了Causality

- **Java**

  - 语言标准的部分概念[10]
    - 对volatile变量的读写和对`Monitor`的Lock/Unlock叫`Synchronization Action`
    - 对volatile变量的写`Synchronizes-With`对该变量的读，对Monitor的Unlock也Synchronizes-With对该Monitor的Lock
    - `Happens-Before`关系(最早由Lamport提出[22])
      - 线程内Program Order靠前的操作x Happens-Before靠后的操作y
      - 如果x Synchronizes-With y则x Happens-Before y
      - Happens-Before关系中的前者看起来先于后者执行
  - 实现Memory Model

按上面标准规定，Volatile Write和Unlock之前的操作应该看起来发生得更早，故之前的操作彼此可以乱序但不能向后越过同步，而之后的操作只要求看起来执行更晚、允许向前乱序越过同步，所以这里的同步操作其实就是前文的Release；对Volatile Read和Lock的分析也类似，它们对应Acquire操作，再加上标准没有要求Atomic Store，最终我们可以弱化的RCsc(写原子性换成Load Other's Store Early && Causality)来实现语言标准中的Memory Model[9]

- **C++**
  - 语言标准[14]
    - std::memory_order

      std::atomic的变量上的操作，可以用以下参数来强制某种顺序或写原子性

      - `std::memory_order_relaxed`：仅保持变量自身读写的相对顺序
      - `std::memory_order_consume`：依赖于该读操作的后续读写，不能往前乱序；另一个线程上std::memory_order_release之前的相关写序列，在std::memory_order_consume同步之后对当前线程可见
      - `std::memory_order_acquire`：之后的读写不能往前乱序；另一个线程上std::memory_order_release之前的写序列，在std::memory_order_acquire同步之后对当前线程可见
      - `std::memory_order_release`：之前的读写不能往后乱序；之前的写序列，对使用std::memory_order_acquire/std::memory_order_consume同步的线程可见
      - `std::memory_order_acq_rel`：两边的读写不能跨过该操作乱序；写序列仅在同步线程之间可见
      - `std::memory_order_seq_cst`：两边的读写不能跨过该操作乱序；写序列的顺序对所有线程相同
  - 实现Memory Model[13]

    C++中的volatile/std::atomic有以下用法，分别对应不同的Memory Model，编译器可以沿用实现这些Memory Model时的手段支持这些用法：

    - `volatile`：只应用于单线程，等同Linux Kernel中的编译器Barrier。*(MSVC中为向后兼容赋予了volatile等同Java的volatile的职责，即读是Acquire写是Release)*
    - `std::memory_order_relaxed` + `Read-Modify-Write`：原子操作，保持变量自身读写的相对顺序
    - `std::memory_order_seq_cst`：相当于WO。因为WO良好的可编程性[1]，该标志是标准库的默认值，一般用它快速实现算法，进一步优化才考虑下面的RCpc
    - `std::memory_order_acq_rel`：相当于弱化的WO(写原子性换成了Load Other's Store Early && Causality，故和std::memory_order_seq_cst相比通不过IRIW测试)
    - `std::memory_order_acquire` + `std::memory_order_release`：相当于RCpc。
    - `std::memory_order_consume` + `std::memory_order_release`：相当于弱化的RCpc。同Linux Kernel中的Data Dependency Barrier + Store Barrier

几个简单的例子

这里用先快速实现、再优化(先WO，再RCpc)，两步走的方式，为几个简单算法加上std::memory_order标志：

- **自旋锁(SpinLock)**
  - Version 1：基于WO(std::memory_order_seq_cst)

```
struct SpinLock {
    void lock() {
        for (;;) {
            while (lock_.load());
            if (!lock_.exchange(true)) break;
        }
    }
    void unlock() {
        lock_.store(false);
    }
    std::atomic<bool> lock_ = {false};
};
```

- 因为WO本来就是最容易编程的Memory Model，同步变量读写彼此不会乱序，临界区内外的操作也不能跨过同步变量读写乱序，故只要以默认标志编写算法即可

- Version 2：基于RCpc(std::memory_order_acquire + std::memory_order_release)

```
struct SpinLock2 {
    void lock() {
        for (;;) {
            while (lock_.load(std::memory_order_relaxed));
            if (!lock_.exchange(true, std::memory_order_acquire)) break;
        }
    }
    void unlock() {
        lock_.store(false, std::memory_order_release);
    }
    std::atomic<bool> lock_ = { false };
};
```

- RCpc是细粒度的Memory Model，程序正确性不容易保证，需要我们逐项自检：
  - 线程安全吗？——线程安全性由改写前的Version 1保证
  - 三种存储器操作的顺序
    - 禁止临界区内s_data读写乱序到外面吗(Data-Race)？——lock的最后一个操作是std::memory_order_acquire，避免了向前乱序；unlock的最后一条操作是std::memory_order_release，避免了向后乱序
    - 允许临界区外p_data读写乱序到里面吗(优化)？——lock中只有std::memory_order_acquire，故允许前面的代码向后乱序；unlock中只有std::memory_order_release，故允许后面的代码向前乱序
    - 禁止sync_var读写间乱序吗(正确性)？——单个std::atomic变量的读写不被乱序
  - 写原子性的要求？——Load Other's Store Early && Non-Causality
- 另一种分析正确性的方法，是根据**Happens-Before**关系推导

读写锁(Readers-Writer Lock)

- Version 1：基于WO

```
struct RWLock {
    void rlock() {
        for (;;) {
            while (writer_.load());

            readers_.fetch_add(1);

            if (!writer_.load()) break;

            readers_.fetch_sub(1);
        }
    }
    void runlock() {
        readers_.fetch_sub(1);
    }
    void wlock() {
        while (writer_.exchange(true));
        while (readers_.load() > 0);
    }
    void wunlock() {
        writer_.store(false);
    }
    std::atomic<bool> writer_ = {false};
    std::atomic<int> readers_ = {0};
};
```

- Version 2：基于RCpc

```cpp
struct RWLock2 {
    void rlock() {
        for (;;) {
            while (writer_.load(std::memory_order_acquire));

            readers_.fetch_add(1, std::memory_order_acquire);

            if (!writer_.load(std::memory_order_acquire)) break;

            readers_.fetch_sub(1, std::memory_order_acquire);
        }
    }
    void runlock() {
        readers_.fetch_sub(1, std::memory_order_release);
    }
    void wlock() {
        while (writer_.exchange(true, std::memory_order_acquire));
        while (readers_.load(std::memory_order_acquire) > 0);
    }
    void wunlock() {
        writer_.store(false, std::memory_order_release);
    }
    std::atomic<bool> writer_ = {false};
    std::atomic<int> readers_ = {0};
};
```

- 自检：
  - 线程安全吗？——由Version 1保证
  - 三种存储器操作的顺序
    - 禁止临界区内s_data读写乱序到外面吗(Data-Race)？——rlock/wlock的最后一个操作是std::memory_order_acquire，runlock/wunlock的最后一条操作是std::memory_order_release
    - 允许临界区外p_data读写乱序到里面吗(优化)？——rlock/wlock中只有std::memory_order_acquire，runlock/wunlock中只有std::memory_order_release
    - 禁止sync_var读写间乱序吗(正确性)？——rlock/wlock内全是std::memory_order_acquire，避免了两个不同地址读写间的乱序
  - 写原子性的要求？——Load Other's Store Early && Non-Causality

- ## 双重检查的单例模式(Double-Checked Locking in Singleton)

  C++11标准的6.7节确保了传统的static局部变量用作Singleton已经线程安全没必要用Double-Checked，这里仅作演示目的

  - Version 1：基于WO

```
template<typename T>
struct Singleton {
    static T* get() {
        T *p = instance_s.load();
        if (p == nullptr) {
            std::lock_guard<std::mutex> guard(mutex_s);
            p = instance_s.load();
            if (p == nullptr) {
                p = new T();
                instance_s.store(p);
            }
        }
        return p;
    }
    static std::atomic<T*> instance_s;
    static std::mutex mutex_s;
};
```

- Version 2：基于弱化的RCpc(std::memory_order_consume + std::memory_order_release)

```
template<typename T>
struct Singleton2 {
    static T* get() {
        T *p = instance_s.load(std::memory_order_consume);
        if (p == nullptr) {
            std::lock_guard<std::mutex> guard(mutex_s);
            p = instance_s.load(std::memory_order_relaxed);
            if (p == nullptr) {
                p = new T();
                instance_s.store(p, std::memory_order_release);
            }
        }
        return p;
    }
    static std::atomic<T*> instance_s;
    static std::mutex mutex_s;
};
```

  - 自检：
    - 线程安全吗？——由Version 1保证
    - 存储器操作顺序
      - 创建对象是在写入instance_s指针前完成的吗？——std::memory_order_release保证了new T()中的操作不会被乱序到后面
      - 访问对象是在读取instance_s指针后完成的吗？(这是典型的Dependent Loads场景)
        - 第1次load：std::memory_order_consume避免了get()返回后的字段访问被乱序到前面(避免编译器激进优化[16]和多处理器的Dependent Loads[4])
        - 第2次load：当p非nullptr返回时，使用std::memory_order_relaxed无法避免get()后的字段访问被乱序到前面(从CPU实际执行的指令序列来看)，但这种乱序是安全

的，因为当前线程的mutex_s上的lock已经与对象创建线程上的unlock同步过，对象字段和instance_s不会再修改

- 写原子性的要求？——Load Other's Store Early && Non-Causality

推荐阅读

数字标号是推荐顺序，其他资料作为参考

1. **Shared Memory Consistency Models: A Tutorial**
   - Computer Architecture: A Quantitative Approach
   - A Primer on Memory Consistency and Cache Coherence
   - Is Parallel Programming Hard, And, If So, What Can You Do About It?
2. **The C++11 Memory Model and GCC**
   - std::memory_order
   - What Every Systems Programmer Should Know About Lockless Concurrency

References

- [1] Shared Memory Consistency Models: A Tutorial
- [2] Computer Architecture: A Quantitative Approach
- [3] A Primer on Memory Consistency and Cache Coherence
- [4] Is Parallel Programming Hard, And, If So, What Can You Do About It?
- [5] Linux Kernel Memory Barriers
- [6] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models
- [7] ARM Cortex-A Series. Programmer's Guide
- [8] JSR 133 (Java Memory Model) FAQ
- [9] The JSR-133 Cookbook for Compiler Writers
- [10] The Java Language Specification, Java SE 9 Edition
- [11] The C# Memory Model in Theory and Practice
- [12] C# Language Specification 5.0
- [13] The C++11 Memory Model and GCC
- [14] std::memory_order
- [15] What Every Systems Programmer Should Know About Lockless Concurrency
- [16] C++ Data-Dependency Ordering: Atomics and Memory Model
- [17] Wiki: Compiler optimizations
- [18] How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs
- [19] Memory access buffering in multiprocessors
- [20] Memory Consistency and Event Ordering in Scalable Shared-Memory
- [21] x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors
- [22] Time, Clocks and the Ordering of Events in a Distributed System