# memory ordering

## 1 Preface

Memory ordering describes the order of accesses to computer memory by a CPU. The term can refer either to the memory ordering generated by the compiler during compile time, or to the memory ordering generated by a CPU during runtime.

This post tries to explain memory ordering in a bottom-up approach, from hardware to software, from abstraction of memory order and memory model to language and application level.

This post covers the following questions

1. How hardware/processors reorder out program (instructions)?
2. What is memory model and memory order?
3. How to understand it on a high level?
4. How to use it (lock-free programming)?

---

## 2 Table of contents

## 3 CPU cache

*A CPU cache is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, L3, L4, etc.).* ...

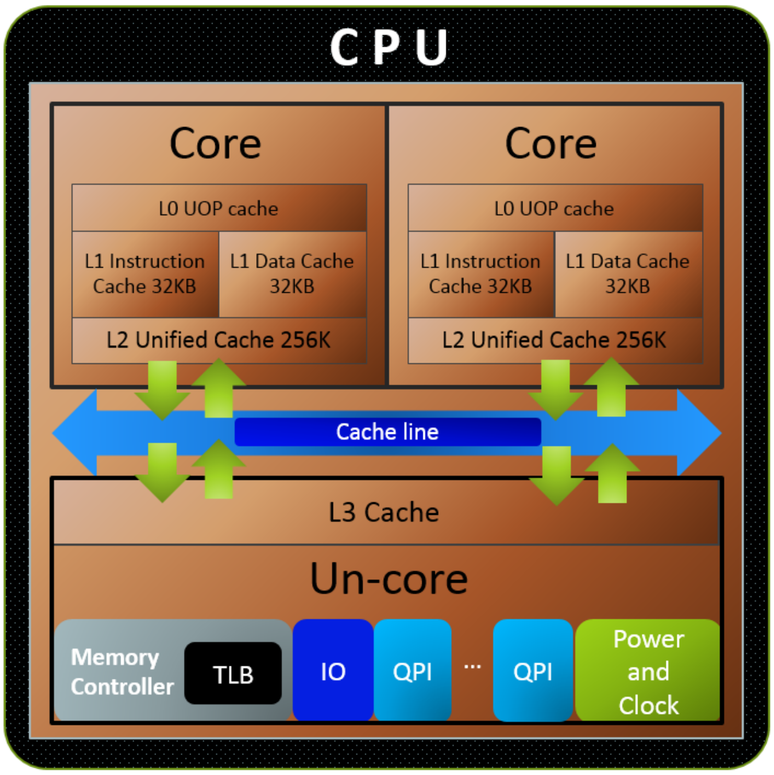


figure 1. general CPU architecture

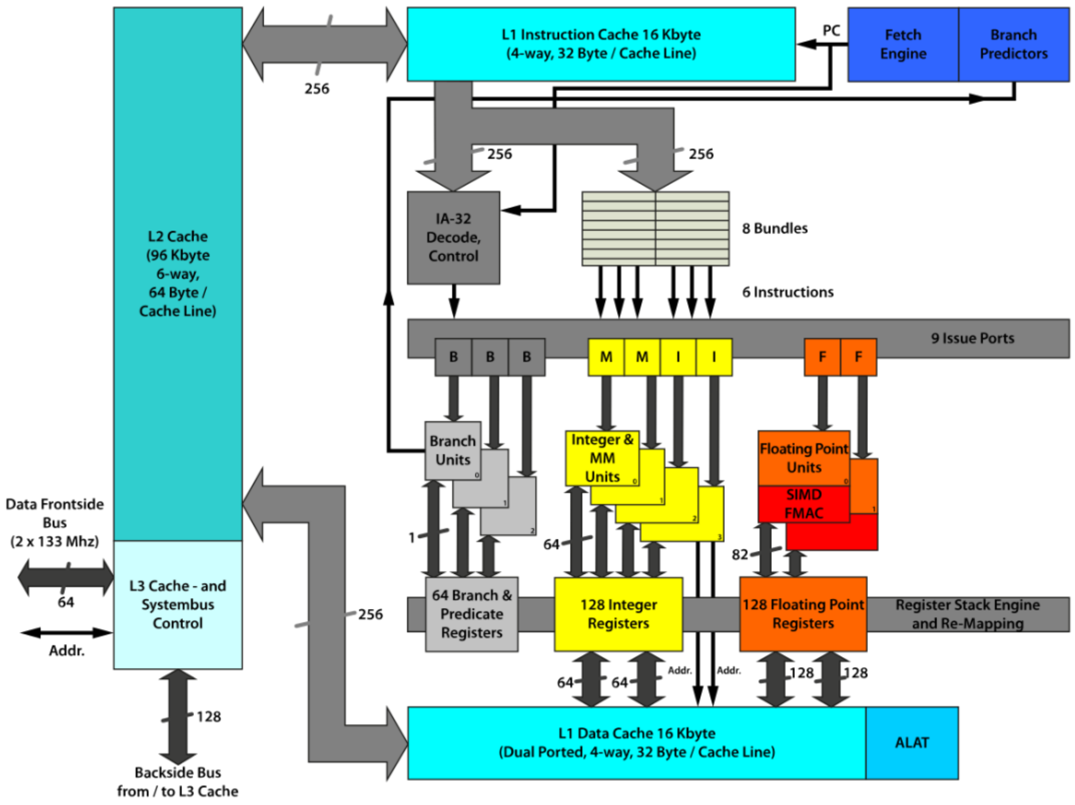

L1 Instruction Cache 16 Kbyte (4-way, 32 Byte / Cache Line)
PC
Fetch Engine
Branch Predictors
256
L2 Cache (96 Kbyte 6-way, 64 Byte / Cache Line)
IA-32 Decode, Control
8 Bundles
6 Instructions
9 Issue Ports
B B B
M M I I
F F
Branch Units
Integer & MM Units
Floating Point Units
SIMD FMAC
Data Frontside Bus (2 x 133 Mhz)
64
Addr.
L3 Cache - and Systembus Control
64 Branch & Predicate Registers
128 Integer Registers
128 Floating Point Registers
Register Stack Engine and Re-Mapping
82
128
Backside Bus from / to L3 Cache
L1 Data Cache 16 Kbyte (Dual Ported, 4-way, 32 Byte / Cache Line)
ALAT

figure 2. Itanium architecture CPU architecture

以上描述摘自 wikipedia，图来自 google，描述和图应该讲的比较清楚对 CPU/CPU cache 有了大概的认识，核心的几个点：

1. L1 分为 data 部分(L1d)和 instruction 部分(L1i)，用途就是字面意思，目前一般来说是 32KB 大小，采用 static memory，速度最快(也最贵)，最接近 CPU 的速度.
2. 大小 `L1 < L2 < L3`，速度 `L1 > L2 > L3`，查找顺序 `L1 > L2 > L3`
3. L3 以及更高 level 的 cache 就是若干个核共享的 cache
4. 现代 CPU 基本都是多核，每个核有自己的 cache(local cache)，增强了性能同时也引入了 额外的**使用复杂性**(cache bouncing 等)
5. 这里没说 Intel 的 hyper threading 技术 就是一个物理核上有俩逻辑核，跟本文没关系不描述了

一般来说，一个核把其处理的数据写到了 L1d 上，其他核就对该数据可见了，这是由 cache coherence 来保证的. 另外，其他核可见是可见了，但是并不保证其他核对这个数据的可见 顺序，比如说有个核写了 2 个数据(a = 0, b = 1)到 L1d，有的核可能先看 先看到 a = 0，然后看到 b = 1；有的核先看到了 b = 1，然后看到 a = 0.

这两个点都是后文要描述的.

## 3.1 Cache line

*Data is transferred between memory and cache in blocks of fixed size, called cache lines or cache blocks. When a cache line is copied from memory into the cache, a cache entry is created. The cache entry will include the copied data as well as the requested memory location (called a tag).*

Cache row entries usually have the following structure:

```
+---------------+---------+-------------------+
|   flag bits   |   tag   |   data block(s)   |
+---------------+---------+-------------------+
```

The data block (cache line) contains the actual data fetched from the main memory.

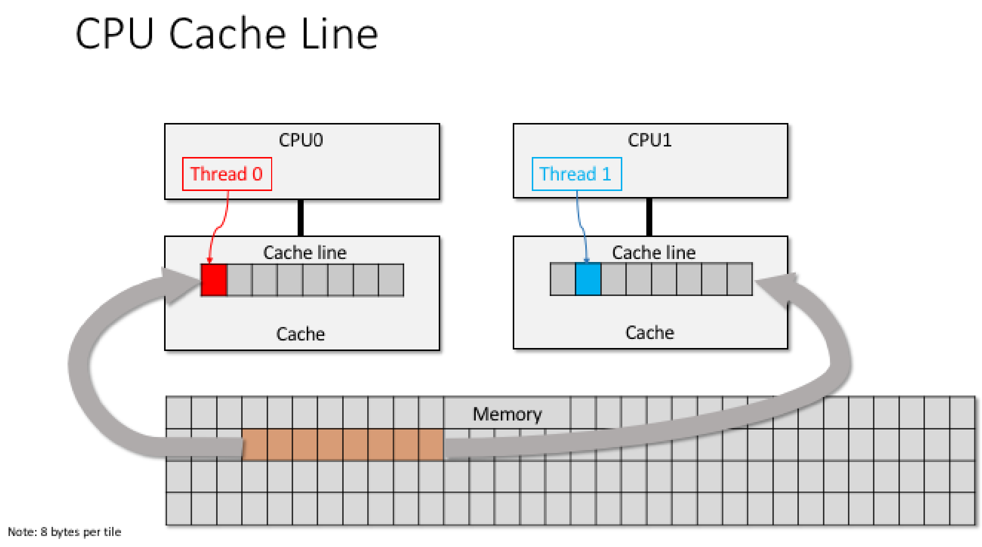


figure 3. CPU cache line

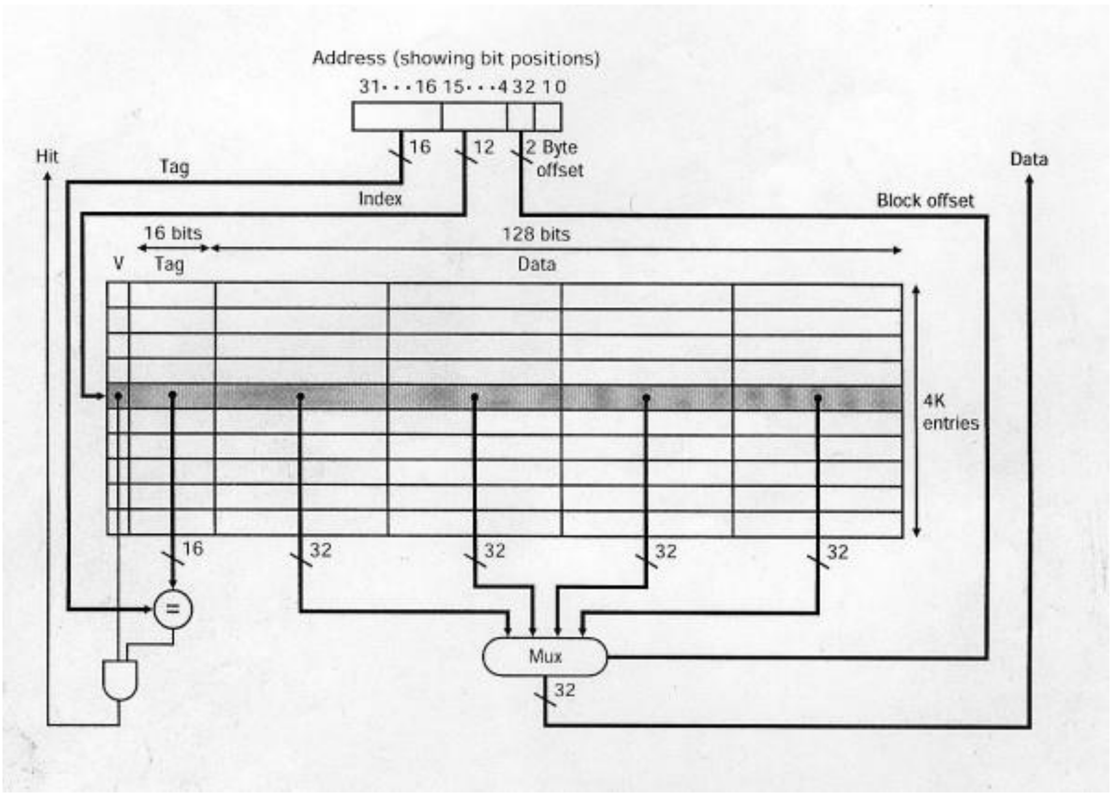


figure 4. CPU cache line data layout

figure 4 shows a block of cache with 4k cache entries with 16bytes(128bit) cache line implemented with “direct mapped cache”, there are other cache implementations such as: set associative cache, column associative cache. Check this video for more info.

在”direct mapped cache”的实现中, cache entry 就是 CPU cache(L1, L2, L3)上一小段 连续的内存, cache entry 从左到又 3 个字段分别为

- valid: 该 entry 是否有效
- tag: cache 里的 key, 一般来说就是 address 的一部分
- data: cache line data, data 部分一般由多个 block 组成, 这样可以只更新 cache line 中 部分数据, 通 address 的末尾若干位(byte offset)来选取(Mux, multiplexer)其中的某 个具体的 data block
- data block: 一个 data block 的大小, 一般和机器的字长是一样的: 32bit or 64-bit

CPU 一个 core 和另外一个 core 或者内存的交互是通过 cache entry 来同步, 也就 是说 CPU 的内存操作粒度是 cache line. 这么做是为了提升效率, 目前来说一般 x86 CPU 的 cache line (data)的大小是 64bytes, 这个也是为什么我们经常说要内存对齐的原因 `__attribute__((align(64)))`或者 c++ keyword `alignas(64)`(since c++11), 如果内存对齐了(按照 cache line size 作为

边界), 对于我们常用的 Intel CPU(Xeon 等) 很多操作其实都变相变成”原子”的了, 因为只需要一次 load/store 就能完成内存的读取/更新.

One more thing, 如果对数据结构做了 alignment, 在一些频繁内存 access 操作的场景内性 能的提升是非常明显的, Herb Sutter 举了这样一个例子.

cache line 的实现是比较有意思的, 如上所述, 这里都是硬件电路, 我们可以根据已有的算 法直接用数字逻辑电路(FPGA 等)来实现一个类似这样的 cache, 这里 是 github 上开源的一个”direct mapped cache”verilog 的实现, 感兴趣可以作为延伸阅读.

### 3.2 Replacement policies

*To make room for the new entry on a cache miss, the cache may have to evict one of the existing entries. The heuristic it uses to choose the entry to evict is called the replacement policy. The fundamental problem with any replacement policy is that it must predict which existing cache entry is least likely to be used in the future. Predicting the future is difficult, so there is no perfect method to choose among the variety of replacement policies available. One popular replacement policy, least-recently used (LRU), replaces the least recently accessed entry.*

*Marking some memory ranges as non-cacheable can improve performance, by avoiding caching of memory regions that are rarely re-accessed. This avoids the overhead of loading something into the cache without having any reuse. Cache entries may also be disabled or locked depending on the context.*

一般来说就是 CPU cache 很小, 比较好的 CPU 总共才几十 MB, 能 mirror(映射)的内存非常有限 , 需要在后续有新 的内存要进到 cache line 按需将之前的一些 cache line 替换掉, 常用的 替换方法有 LRU 等.

### 3.3 Write policies

*If data is written to the cache, at some point it must also be written to main memory; the timing of this write is known as the write policy. In a write-through cache, every write to the cache causes a write to main memory. Alternatively, in a write-back or copy-back cache, writes are not immediately mirrored to the main memory, and the cache instead tracks which locations have been written over, marking them as dirty. The data in these locations is written back to the main memory only when that data is evicted from the cache. For this reason, a read miss in a write-back cache may sometimes require two memory accesses to service: one to first write the dirty location to main memory, and then another to read the new location from memory. Also, a write to a main memory location that is not yet mapped in a write-back cache* *may evict an already dirty location, thereby freeing that cache space for the new memory location.*

因为 CPU 各个 core 和内存之间存在 cache，这里就涉及到 cache line 变更之后何时 写到内存里，简单的办法是更新 cache line 的时候就直接往内存里更新对应数据(不考虑 和其他 core 的 cache 的交互，后续有介绍)，这样最简单但是效果不一定最好，另外有一些 方法比如 `write-back`，`copy-back`，选个合适的时机(比如说 cache line 淘汰，有新的 read 等)"异步"将数据写回内存. 具体的各个厂商实现也有将上述写回内存策略混合使用的.

## 3.4 Cache coherence

Cache coherence is the uniformity of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

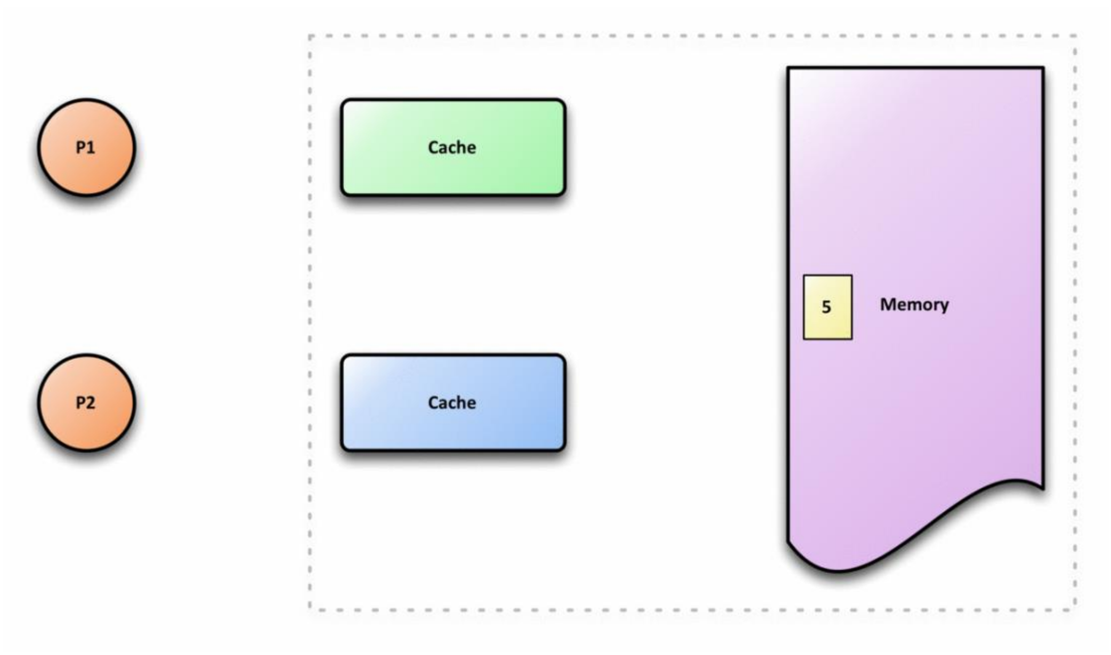


figure 5. cache coherent

figure 5, from wikipedia，展示了 cache coherence 的大概过程 cache coherence 就是讲 cache line 同步给各个核以及内存的交互过程(协同/协调？具体不知道怎么翻译), CPU 的 runtime 指令重排以及 memory ordering model 跟这个实现很有关系

cache coherence 是一个非常复杂的过程, intel 在其公布的文档中很少有说道他们是如何实 现的，但是 Intel 的 CPU 就是能保证一个核的更新了 cache line 之后

另外的核以及内存对该 cache line 的值都能达到一致的状态，这个有点类似于分布式中的一致性.

说到这个 cache coherence，一般人都会有以下两个疑问:

1. *Two cores try to modify the same cache line "simultaneously". Who will win? Can both lose? How does intel implementatio handle this case?*
2. *One core try to read from a memory location which is not in cache while another one has exclusive ownership of a cache line with this memory location and try to write some value into it (simultaneously). Who will win? The cache line state will first transfered to a Shared state and then invalidated or Modified and then Shared?*

这个东西很难，但是看起来这两个问题对 Intel 来说很轻松就做到了？这是一个专家对以上问题的解答(how intel cache coherence works)

*Intel does not document the details of its coherence protocols, but the ordering model is described in some detail in Section 8.2 of Volume 3 of the Intel Architectures Software Developer's Manual (Intel document 325384).*

*The details of the implementation of the coherence protocol (and its associated ordering model) differ across processor models, and can differ based on configuration settings (e.g., Home Snoop vs Early Snoop on Haswell EP, or 1-socket vs 2-socket vs 4-socket configurations). In addition, some coherence transactions can be handled in different ways, with the choice made dynamically at run time.*

*The most interesting details usually pop up in the Uncore Performance Monitoring Guides for the various processor models. Understanding the implications of these details requires significant experience in microarchitecture and microbenchmarking, though there are many aspects of Intel's implementations that are probably too hidden to be fully understood by anyone who does not have access to the actual processor RTL (or whatever high-level-design language Intel uses these days).*

*The short answer to #1 is that coherence processing is serialized at the various coherence agents. For example, if two cores execute a store instruction in the same cycle and both miss in their L1 and L2 caches, then the transaction will go to the L3 slice (or CHA in some processors) responsible for that address, which will process the incoming requests sequentially. One or the other will "win" and will be granted exclusive access to the cache line to perform the store. During this period, the request from the "losing" core will be stalled or rejected, until eventually the first core completes its coherence transaction and the second core's transaction is allowed to proceed. There are many ways to implement the details, and Intel almost certainly uses different detailed approaches in different processor models.*

*The short answer to #2 is that also that the transactions will be serialized. Either the owning processor will complete the store, then the line will be transferred to the reading processor, or the cache line will be transferred away from the owning processor to the reading processor first, then returned to the (original) owning processor to complete the store. Again, there are many ways to implement the low-level details, and Intel processors implement several different approaches. If I am reading between the lines correctly, recent Intel chips will use different transactions for this scenario depending on how frequently it happens to a particular address. (E.g., See the description of the HitMe cache in the Xeon E5/E7 v3 processor family uncore performance monitoring guide, document 331051.)*

*"Dr. Bandwidth"*

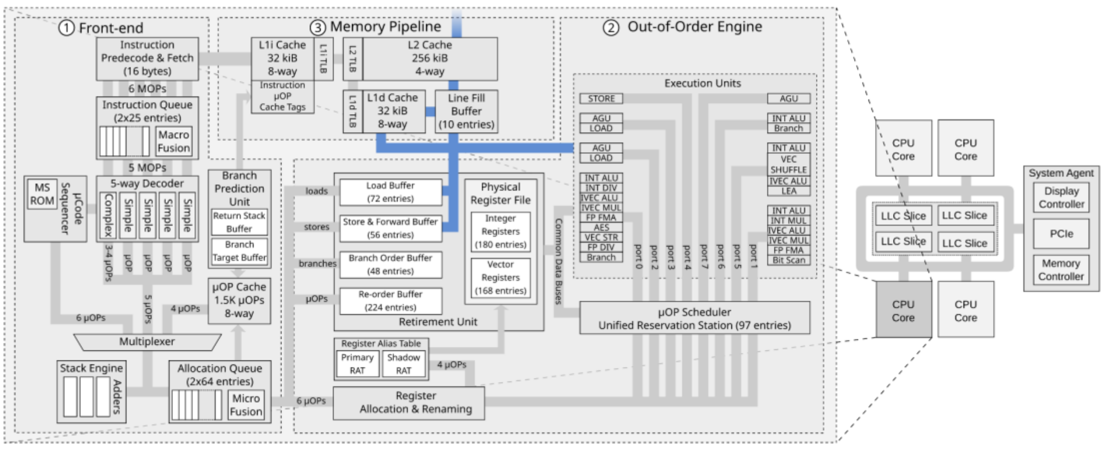
cache coherence 解决了核之间数据可见性以及顺序的问题，本质上可以把 cache 当做内存 系统的一部分，有没有 cache 对各个 CPU 来说都是一样的，只要写到了 cache(所以可以有很多级 cache)，其他核就可以看到该数据，并且顺序是确定的 – 保证顺序一致性.

MESI 是一种 cache coherence 协议，可以搜索关键词获取更多信息，这里不展开阐述.

对内存可见性和顺序有影响的部件是接下来要介绍的 store buffer.

## 3.5 Store buffer

在 CPU 上除了除了 L1 cache，其实还有更靠近 CPU 的跟数据相关的"cache"，一个是 load buffer，一个是 store&forward buffer(一般就叫做 store buffer)．注意看下图中蓝线连接的部分.



这两个 buffer 是上图中② out of order engine 中的一部分，顾名思义就是跟乱序执行有关 系.

数据读取(load buffer)跟不影响数据对外的可见性，但是 store buffer 会，数据从 CPU 的 store buffer 出来到 cache，上其他核就可以看到该数据.

store buffer 类似于一个队列，但是大小是按照 entry 计算不是按照 bits 计算的. 上图的例子是 56 entries，一般来说 store buffer 的 size 是 load buffer 的 2/3，因为大 多数程序读别写多.

开了超线程(hyper-threading), store buffer 这些资源就要等分成两份，分别给两个逻辑 核使用.

*The following resources are shared between two threads running in the same core:*

- *Cache*
- *Branch prediction resources*
- *Instruction fetch and decoding*
- *Execution units*

*Hyperthreading has no advantage if any of these resources is a limiting factor for the speed.*

CPU 操作数据变更之后并不会立即写到 L1d 上，而是先写到 store buffer 上，然后尽快写到 L1d cache，比如如下代码

```
int sum = 0;
for (int i = 0; i < 10086; ++i) {
  sum += i;
}
```

`sum` 完成一次赋值之后不会立即写到 L1d 上，可能会在 store buffer 上等一段时间，然后 第二次又赋值之后才从 store buffer 刷到 L1d 上. 这样做是为了提高效率，预测器(speculator)，把两个相加语句先执行了，然后再写到 L1d 里，这样就好像把 +1 +2 两条的结果直接合并成+3 一次写到 L1d 里，节省了写 cache 的时间.

根据 store buffer 的特点，再举一个例子, store buffer 会引起”乱序”的问题

```
// global ini
int a = 0;
int b = 0;

// thread 1                    |          // thread 2
```

```
t1:                           |          t2:
a = 1;                        |          b = 1;
if (b == 0) {                 |           if (a == 0) {
  // do something             |             // do something
} else {                      |           } else {
  goto t1; // retry           |             goto t2; // retry
}                             |          }
```

上述代码执行完之后，两个线程都能同时走到 “do something” 的逻辑里，因为 thread 1 在写 a = 1 之后，数据在 store buffer 里，对 thread 2 其实是不可见的，这个时候 thread 2 看到的还是 a == 0，同理, thread 2 写完 b = 1 会在 store buffer 里停留一段时间, thread 1 也看到 b == 0，这样两个线程都认为自己可以进入到”do something” 的逻辑里.

这个其实是著名的”Peterson’s and Dekker’s algorithm”(互斥锁)，上述例子也阐述了这个算法在现代 CPU 架构上这么做是不可行的.

需要让该算法正确运行，把 a b 替换成 atomic 即可，因为 atomic 引入了 memory model 的 可见性顺序保证，能够保证 thread 1 的数据写了之后把数据从 store buffer 刷出到 cache 并 且同步到其他核, thread 2 看到的是线性一致的最新的数据，后文会详细说明这其中的原因 和原理.

### 3.6 Conclusion of CPU cache

CPU cache 加快了执行的速度，但是也引入了额外的使用问题, CPU cache 本身的实现逻辑就 很复杂，即使实现了 cache coherence，由于 store buffer 的存在，只要有多个核共同工作 的环境下还 是会有各个核对内存可见顺序的问题，再加上编译器和 CPU 的优化，问题就显得 更加复杂了，我们使用高级语言所写的程序到真正执行的时候也许已经不是我们想象的那样 了.

接下来，本文将阐述数据可见性顺序的问题以及如何解决.

# 4 Instruction reordering

在说 memory order/barrier/fence 之前，我们需要知道一个概念就是 instruction reordering(指令重排)，指令重排主要有两种，一种是 compiler 在 compile-time 产生的指令 重排，一种是 CPU 在 rumtime 产生的指令重排，两者都是为了提高运行效率:

1. compile-time: 在编译器重排编译生成指令达到减少使用寄存器 load/store 的次数，比 如编译器发现有个寄存器里的值可以先写回到内存里再 over write 该寄存器的值作它用

2. runtime: 在运行时预测(speculate)在同个 instruction stream 里哪些指令可以先执行，同时减少类似重复 load/store 内存或者寄存器，比如说对分支的 speculative execution

compile-time reordering 也可以叫做 compiler reordering，就是静态的 reordering，runtime reordering 也可以叫做 CPU reordering，就是动态的 reordering，下文不在另行 解释.

但是无论 compile-time 还是 runtime 的指令重排都不是随便重排的，是要有遵守一定的规则，总结起来就是:

*Thou shalt not modify the behavior of a single-threaded program.*

这好像是一句名言，还包含古词... 没找到详细出处，意思就是不管如何重排，都不能影响其重排后在单线程里跑出来的结果的正确性，就是重排 后的结果和重排前的结果一样，这里就要涉及到数据的依赖问题.

In my point of view, **memory order 的问题就是因为指令重排引起的**，指令重排导致 原来的内存**可见顺序**发生了变化，在单线程执行起来的时候是没有问题的，但是放到 多核/多线程执行的时候就出现问题了，为了效率引入的额外复杂逻辑的的弊端就出现了.

观察如下代码

```
int msg = 0;
int ready = 0;

// thread 1
void foo() {
  msg = 10086;
  ready = 1;
}

// thread 2
void bar() {
  if (ready == 1) {
    // output may be 0, un expected
    std::cout << msg;
  }
}
```

code 1. msg and ready

thread 1 单线程执行 `foo()`先给 a 赋值和先给 b 赋值都是最终的结果都是正确的，符合重排的最 基本的 principle，所以在不加任何限制的情况下，无论是 compiler 生成的汇编或者 CPU 在直 线 instruction stream 的时候都可以自由地将 ready 先于 msg 赋值.

但是，引入(thread 2)多线程之后，由于业务逻辑上需要有数据的依赖关系, ready 之后才 能 cout msg 但是 thread 2 并不知道 ready 和 msg 有什么顺序关系，只负责自己的 `bar()`执 行即可，在 thread 2 看起来自己的执行也是正确的，但这个时候运行起来就有”bug”了.

这个时候就需要有另外的技术来解决的问题，就是我们要说的 memory order/barrier/fence. 这里提到 3 个词 memory order, memory barrier, memory fence，其 实本质上说的都是一个问题(内存可见性)，后文就不做区分了.

## 4.1 Compile-time reordering

接下来具体观察一下 compile-time reordering

```
// cpp code                      | ;asm code
                                |
int a = 0;                       |
int b = 0;                       |
                                |
void foo() {                     |   foo():
  a = b + 1;                     |          mov     eax, DWORD PTR
b[rip]
  b = 5;                        |          mov     DWORD PTR
b[rip], 5
}                               |          add     eax, 1
                                |          mov     DWORD PTR
a[rip], eax
                                |          ret
```

code 2a. compile-time reordering, `a = b + 1`

以上代码使用 x86-64 gcc 8.2 编译，编译选项`-O2`，其他 architecture 的 CPU 代码也许不一 样，在我理解，编译器的这个”优化”看起来是很合理的，把同个内存

地址的都先处理了，以 防后后续还 有对该内存的处理从而需要额外使用寄存器来操作该内存(之前的寄存器可能被 覆盖了).

我们可以通过强制约束编译器来 get rid of this kind of reordering. 在两个赋值语句之 间添加一个"compile-time memory order constraint".

```
// cpp code                         | ;asm code
                                   |
int a = 0;                          |
int b = 0;                          |
                                   |
void foo() {                        | foo():
  a = b + 1;                        |        mov     eax, DWORD
PTR b[rip]
  std::atomic_thread_fence(           |        add     eax, 1
    std::memory_order_release);       |        mov     DWORD PTR
a[rip], eax
  b = 5;                             |        mov     DWORD PTR
b[rip], 0
}                                   |        ret
```

code 2b. compile-time reordering constraint, a = b + 1

可以看到在我们添加

`std::atomic_thread_fence(std::memory_order_release)`限制之后，生成两个赋值的汇编代码就按照我们的 cpp 代码顺序进行了, (至于为什么是 `release` 而不是其他 `acquire` 或者 `seq_cst`，后文有相应的说明).

这里想说的是, compile-time reordering 是我们能够控制的，一般情况下编译器的优化 默认会进行一些 reordering，如果没有说明特殊的需求，不需要显式限制编译器的优化.

除了 code2a 展示的优化 还有一些 redundant assignment 优化

```
// cpp code                | ; asm code
int a = 0;                 |
int b = 0;                 |
void foo() {               | foo():
  a = 1;                   |        mov     DWORD PTR b[rip], 2
  b = 2;                   |        mov     DWORD PTR a[rip], 3
  a = 3;                   |        ret
}
```

使用 g++8.2 -O1, 上述优化意图就很明显了, 不再赘述. (使用 volatile 可以防止这种优 化)

Herb Sutter 在这个 talk(atomic Weapons 2 of 2) 里提到了一些编译器在做 reordering 优化的时候的一些考虑, 讲解了一些有意思的编译器考 虑 reordering 优化的 case.

目前还没找到更详细的 compiler reordering 策略相关的资料, 这是一个非常复杂编译器优 化方向, 也许可以通过学习 gcc, clang 等编译器来进行进一步了解, 如果有读者有相应的资 料可以 share 一下.

### 4.2 Runtime reordering

不同的 CPU 架构(厂商)runtime reordering 的策略也不一样, 这里涉及另外一个 topic, 就是 CPU 的 hardware memory model, 这个文章有比较详细的介绍, 这里结合自己的经验和理解只做一下总结:

1. 不同的 CPU 架构的 hardware memory model 是不一样的, 就是指令重排的策略/限制不一样
2. 大概分为两大类型, weak & strong
    1. weak hardware memory model: ARM, Itanium, PowerPC, 对指令重排没有太多"限制", 我们前边提到的 CPU cache 的复杂架构和实现对增加内存的可见性(顺序)的复杂度贡献 还是很大的, 但是正是这些比较少的限制, 可以使软件开发人员有更多的选择来实现性能优化.
    2. strong hardware memory model: x86-64 family (Intel and AMD), 限制了很多类型的指令重排

*A strong hardware memory model is one in which every machine instruction comes implicitly with acquire and release semantics. As a result, when one CPU core performs a sequence of writes, every other CPU core sees those values change in the same order that they were written.*

一个核有一个 instruction stream, 假设其中有 n 个对内存 write 操作, 如果执行到第 k 个 write 指令时, 前 k-1 个 write 都能被其他核观察到, 并且其他核观察到的这个 k 个 write 的顺序和在本核上的顺序是一致的. 要保证这个是有一定的性能损耗, 因为有 CPU cache 的存在, 增加了这个可见性的难度, strong model 在一定程度上也限制了优化.

3. 即使是同为 weak/strong，本身差别也很多，因为指令集千差万别，同个指令集的指令组 合起来就有数不清的可能性，所以各个 CPU 架构的策略都是不一样
4. *但是*，不管是哪种架构的 CPU, CPU 指令重排都遵循一定的规则，这些规则就写在 了各个 CPU 的使用手册里，这些手册是系统(OS)开发人员必须读的.
5. 对于 Intel 等 x86 系列的 CPU，对应厂商的思路是，只要我单条指令执行的够快，就不用太 担心优化的问题.

上边总结 2.2 中提到一个概念”acquire and release semantics”，会在后续章节展开描述.

---

### 4.2.1 Gotcha! Runtime reordering

接下来我们需要证明一下 runtime reordering 是真实存在的，我们要清楚一个事实: runtime reordering 比较难捕获，毕竟不能直接静态分析，一定要实际运行，而运行期间还 不能每次必现，因为各个时刻 CPU 的 instruction stream 都是不一样的.

以代码片段 code 1 为例，可以不断起线程来检测，改写成如下代码

```
void test() {
  for (;;) {
    int msg = 0;
    int ready = 0;
    // thread 1
    std::thread t1([&msg, &ready] {
        msg = 10086;
        ready = 1;
      }
    );

    // thread 2
    std::thread t2([&msg, &ready] {
        if (ready == 1 && msg == 0) {
          std::cout << "gocha" << std::endl;
        }
      }
    );
    t1.join();
    t2.join();
  }
}
```

code 3. simulate runtime reordering with 2 threads

**但，遗憾的是，这段代码在 Intel x86-64 上是捕获不到 runtime reordering 的**，因为 Intel x86-64 限制 了这种重排，就是说 CPU 并不会对这两句进行重排.

如果是在其他类型的 CPU 上执行上述代码，也比较难捕获到 runtime reordering，因为

```
msg = 10086;
// only 1 instruction after last store/assignment
ready = 1;
```

这两个赋值之间最小距离是 1 个指令，在运行期很难被另外一个核上的线程(thread2)观察的 到中间状态，所以我们需要使用另外的方式来观察

我们利用赋值顺序的结果来观察，不直接观察中间状态，"抓拍"到 runtime reordering 的机 会就会大很多

一个典型的例子，也可以再 x86-64 平台上跑

```
// the static instruction stream (compiled object)

  thread1                          thread2

mov [_X], 1                      mov [_Y], 1
mov r1, [_Y]                     mov r2, [_X]
```

X, Y 是内存地址, r1, r2 是寄存器，两个线程执行的汇编代码过程类似. 第一句是一个 store 操作第二句是一个 load 操作，两个汇编语句操作的内存地址都不一样.

以上汇编在本各自的线程(核)看来先执行 load 或者 store 操作都是可以的，都不违反指令重 排的基本原则.

X, Y are both initially zero, if CPU instruction reordering occurs in both threads, that is to say `r1 == 0 && r2 == 0`, the execution order is

```
  thread1                          thread2
```

```
mov r1, [_Y]                mov r2, [_X]
mov [_X], 1                 mov [_Y], 1
```

**for Intel only store-load reordering is allowed when there is no memory fence to prevent reordering**, I will talk this later in Intel's reordering specification.

可以用如下代码来进行模拟, click here for full source code. 使用 gcc 4.8.2 以上版本或者对应的 clang 版本, 编译选项如下.

```
g++ -std=c++11 -lpthread
```

如下是模拟 thread1 的过程, 中间 `asm volatile` 部分是显式限制编译器产生的汇编代码将 `r1 = Y` 放到 `X = 1` 之前执行(限制编译器的指令重排), 在 x86-64 架构下 `asm volatile` 这个可以使用 c++11 语法 `std::atmoic_thread_fence(std::memory_order_relase)` 代替.

```
int X, Y;
int r1, r2;

void thread1_func() {
  std::mt19937 rng(std::random_device("/dev/random")());
  std::uniform_int_distribution<int> random(1, 100000000);
  for (;;) {
    sem_wait(&begin_sema1);  // Wait for signal
    // Random delay is necessary to increase the posibility of
capaturing CPU
    // reordering, this demo is still working without this
random delay
    while (random(rng) % 8 != 0) {}

    X = 1;
    // Prevent compiler reordering explicitly
    asm volatile("" ::: "memory");
    r1 = Y;

    sem_post(&end_sema);  // Notify transaction complete
  }
};
```

code 4. thread1_func

thread2 的过程和 thread1 类似，把 1 换成 2，把 X 换成 Y 即可，不在此赘述.

main 函数如下，主要是模拟两个线程在两个核上同时执行,

```
int main(void) {
  sem_init(&begin_sema1, 0, 0);
  sem_init(&begin_sema2, 0, 0);
  sem_init(&end_sema, 0, 0);

  std::thread thread1([] { thread1_func(nullptr); });
  std::thread thread2([] { thread2_func(nullptr); });

  // Repeat the experiment ad infinitum
  int detected = 0;
  for (int iterations = 1; ; ++iterations) {
    // Reset X and Y
    X = 0;
    Y = 0;
    // Signal both threads
    sem_post(&begin_sema1);
    sem_post(&begin_sema2);
    // Wait for both threads
    sem_wait(&end_sema);
    sem_wait(&end_sema);
    // Check if there was a simultaneous reorder
    if (r1 == 0 && r2 == 0) {
      std::cout << "gotcha! "
        << ++detected << " reorders detected after " <<
iterations
        << " iterations" << std::endl;
    }
  }
  return 0;  // Never returns
}
```

code 5. main function of capturing CPU reordering

在 `Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz` 的机器上运行输出结果如下，大概有 1%的 概率能捕获到 CPU reordering.

```
...
gotcha! 5505 reorders detected after 563904 iterations
gotcha! 5506 reorders detected after 565918 iterations
gotcha! 5507 reorders detected after 569906 iterations
gotcha! 5508 reorders detected after 571118 iterations
gotcha! 5509 reorders detected after 573014 iterations
gotcha! 5510 reorders detected after 573665 iterations
gotcha! 5511 reorders detected after 573871 iterations
gotcha! 5512 reorders detected after 574064 iterations
gotcha! 5513 reorders detected after 575861 iterations
gotcha! 5514 reorders detected after 576289 iterations
gotcha! 5515 reorders detected after 578107 iterations
gotcha! 5516 reorders detected after 580264 iterations
...
```

---

#### 4.2.2 Intel x86-64 family reordering specification

**This section first introduces what kind of runtime reordering is allowed and what is not, and then introduces some critical for core/cache synchronization.**

Most of the content is copied from Intel's developer manual volume 3 §8.2. I put it here not only for a reference purpose but also for a demo of hardware memory model. x86-64 is the most popular processor, it can be a representative for other CPUs.

---

Different CPU architectures have different hardware memory models, the modern Intel CPU families: Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6, use so called strong hardware memory order model.

Most of the following specification is copied from Intel's specification.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (Note the memory-ordering principles for single-processor and multiple- processor systems are written from the perspective of software executing on the processor, where the term "processor" refers to a logical processor. For example, a physical processor supporting multiple cores and/or Intel Hyper-Threading Technology is treated as a multi-processor systems.):

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions:
    - streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and
    - string operations (see §8.2.4.1).
- No write to memory may be reordered with an execution of the CLFLUSH instruction; a write may be reordered with an execution of the CLFLUSHOPT instruction that flushes a cache line other than the one being written.1 Executions of the CLFLUSH instruction are not reordered with each other. Executions of CLFLUSHOPT that access different cache lines may be reordered with each other. An execution of CLFLUSHOPT may be reordered with an execution of CLFLUSH that accesses a different cache line.
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
- Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.
- Reads cannot pass earlier LFENCE and MFENCE instructions.
- Writes and executions of CLFLUSH and CLFLUSHOPT cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes or executions of CLFLUSH and CLFLUSHOPT.
- MFENCE instructions cannot pass earlier reads, writes, or executions of CLFLUSH and CLFLUSHOPT.

In a multiple-processor system, the following ordering principles apply:

- Individual processors use the same ordering principles as in a single-processor system.
- Writes by a single processor are observed in the same order by all processors.
- Writes from an individual processor are NOT ordered with respect to the writes from other processors.
- Memory ordering obeys causality (memory ordering respects transitive visibility).

- Any two stores are seen in a consistent order by processors other than those performing the stores
- Locked instructions have a total order.

Check §8.2.3 “Examples Illustrating the Memory-Ordering Principles” of Intel Software Developer Manuals Combined Volumes: 3a, 3b, 3c and 3d - System programming guide for detailed instruction reordering examples.

In summary, for simple store/load instructions, I haven’t talked about string instructions yet, the only CPU reordering case is:

Loads May Be Reordered with Earlier Stores to Different Locations

```
Processor 0                      |        Processor 1
                                 |
mov [_x], 1                      |       mov [_y], 1
mov r1, [_y]                     |       mov r2, [_x]

Initially x = y = 0
r1 = 0 and r2 = 0 is allowed
```

code 6. case that CPU may reorder the instructions

#### 4.2.2.1 Fence instruction

The SFENCE, LFENCE, and MFENCE instructions provide a performance-efficient way of ensuring load and store memory ordering between routines that produce weakly-ordered results and routines that consume that data. The functions of these instructions are as follows:

- SFENCE — Serializes all store (write) operations that occurred prior to the SFENCE instruction in the program instruction stream, but does not affect load operations.
- LFENCE — Serializes all load (read) operations that occurred prior to the LFENCE instruction in the program instruction stream, but does not affect store operations.
- MFENCE — Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

That is to say, we can use one of the above fence instructions to explicitly prevent the CPU from reordering.

```
Processor 0                        |           Processor 1
```

```
                                     |
mov [_x], 1                           |         mov [_y], 1
mfence       ;;;full memory fence |
mfence      ;;;full memory fence
mov r1, [_y]                          |         mov r2, [_x]

Initially x = y = 0
r1 = 0 and r2 = 0 is impossible
```

code 7. introduce `mfence` to prevent CPU reordering

The above code has introduce a full memory fence, which is relative heavy operation, of code 6,

#### 4.2.2.2 Locked instruction

Locked instructions will lock the bus and synchronize all the caches of CPU,

*Intel 64 and IA-32 processors provide a LOCK# signal that is asserted automatically during certain critical memory operations to lock the system bus or equivalent link. While this output signal is asserted, requests from other processors or bus agents for control of the bus are blocked.*

which is usually a read-modify-write(RMW) does, the common case is a `compare-and-swap` operation

```
// cpp code                            | ; asm code
std::atomic<int> ai = { 0 };            |
                                      |
void cas() {                           | cas():
  int a;                               |    xor     eax, eax
  ai.compare_exchange_weak(a, 10086,    |    mov     edx, 10086
      std::memory_order_relaxed);       |    lock cmpxchg  DWORD
PTR ai[rip], edx
}                                     |    ret
```

§8.1 "LOCKED ATOMIC OPERATIONS" of Intel's developer manual volume 3 has described how does the locked instruction work and how the atomic operations are guaranteed.

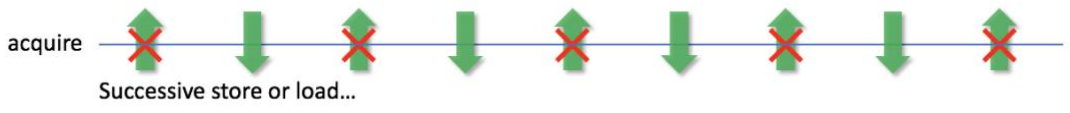
## 5 Acquire and Release semantics

先总结什么是 Acquire and Release semantics:

- acquire semantics，硬件层面就是保证在此之后的所有指令在执行的时候都不能重排到这 个指令之前，软件层面在此之后的所有操作都不能重排到这个语义(fence)之前
- release semantics，硬件层面就是保证在此之前的所有 store 都已经”release”可见，软件层面在此之前的所有操作都不能重排到这个语义(fence)之后

这个概念是对各个核对内存的变更在其他核的可见性，这个也是和 CPU cache 的复杂实现是息息相关的，因为各个厂商实现不一样，但是这些内存的可见顺序的行为是可以通过这个抽象出来的概念来描述的. **这个概念不仅仅是对 CPU reordering 的约束也是对 compiler reordering 的约束.**

*Acquire semantics is a property that can only apply to operations that read from shared memory, whether they are read-modify-write operations or plain loads. The operation is then considered a read-acquire. Acquire semantics prevent memory reordering of the read-acquire with any read or write operation that follows it in program order.*



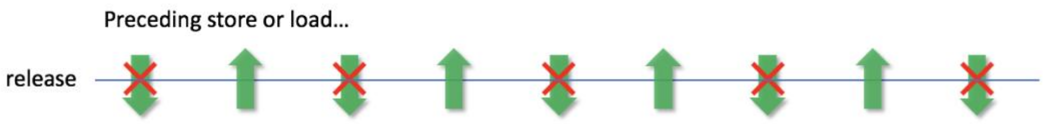
`acquire` 和 `release` 是配对的，硬件层面，这个语义(也许是一条 CPU instruction)在执行 这个”指令”之后的所有内存的 load 操作都看到的 其他核”最新”的变更(depends on memory model of the CPU)，也就是说在这个”指令”之后的所有对的指令都不能 reorder 到这个”指令”之前. 同样的，在 ARM 上这个”指令”是 `dmb`，在 PowerPC 上是 `lwsync`，在 x86-64 上不需要单独的指令.

软件层面，如果我们所编写代码包含了这个语义，编译器在翻译成汇编的时候要在优化的时 候考虑程序字面的要求意思，确保

*prevent memory reordering of the read-acquire with any read or write operation that follows it in program order.*

能够满足.

*Release semantics is a property that can only apply to operations that write to shared memory, whether they are read-modify-write operations or plain stores. The operation is then considered a write-release. Release semantics prevent memory reordering of the write-release with any read or write operation that precedes it in program order.*



`release samentic`和字面意思很像. 硬件层面, 这个语义(也许是一条 CPU instruction)在执行这个 "指令"之后, 发生在这个"指令"之前(happens-before)的所有内存的 store/load 操作在所有 核都可见, 更直观的解释就是在这个"指令", 之后的所有指令不能 reorder 到这个"指令" 之前. 在 ARM 上这个"指令"是 `dmb`, 在 PowerPC 上是 `lwsync`, 在 x86-64 的 CPU 上不需要 单独 的一个"指令"完成这个事情, 因为 x86-64 的 strong hardware memory model 类型的处 理器. 软件层面, 如果我们所编写代码包含了这个语义, 编译器在翻译成汇编的时候要在优化的时 候考虑程序字面的要求意思, 确保

*prevent memory reordering of the write-release with any read or write operation that precedes it in program order*

能够满足.

**memory order 有软件层面的也有硬件层面的 order**, 对应的就是 compile-time reordering 和 runtime reordering.

这也稍微从更抽象的层面解释了为什么用 C++11 标准使用说明里说 `std::memory_order_relase` 和 `std::memory_order_acquire` 要配对使用了, 如果不配对 使用, 其实只限制了其中一部分 reordering 的行为, 另外一部分 reordering 的行为未限制, 不会达到期望的 reordering 效果.

关于 `acquire and release semantics`, 还可以参考这个文章 以及这个视频"Herb Sutter - atomic Weapons 1 of 2" 获取更多地对于这个概念的讲解和讨论.

## 5.1 Sequential Consistency

讲完 `Acquire and Release semantics`，就需要详细讲一下”Sequential Consistency” (顺序一致)这个概念，这个概念是 Lamport 大神在 1979 年提出的，一般在分布式系统里用的比较多，但是一个 multi-core CPU+内存何尝不是一个非常典型的 multiprocessing system 呢？

*Leslie Lamport, 1979, who defined that a multiprocessing system had sequential consistency if:*

*“…the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”*

*This definition was one of the first statements on what constituted “correct” behavior for a symmetric multiprocessing (SMP) system. It has been restated in various forms, among them the following:*

*Sequential consistency requires that a shared memory multiprocessor appear to be a multiprogramming uniprocessor system to any program running on it.*

*Sequential consistency requires that:*

1. *All instructions are executed in order, result in a single total order*
2. *Every write operation becomes instantaneously visible throughout the system*

如 figure 6 中所示(纵向是时间)．考虑有两个线程，线程 1 需要执行 AB，线程 2 需要执行 CD，如果 ABCD 每个操作都是**原子**的，在并行系统中，如果这个系统是保持 sequential consistency 的，那么最终的执行顺序是类似左边中的一种，但是绝对不会是类似于最右边 的那种，并且对于每个线程，最终看到的执行顺序也只有一种–各个线程(核)达成一致.

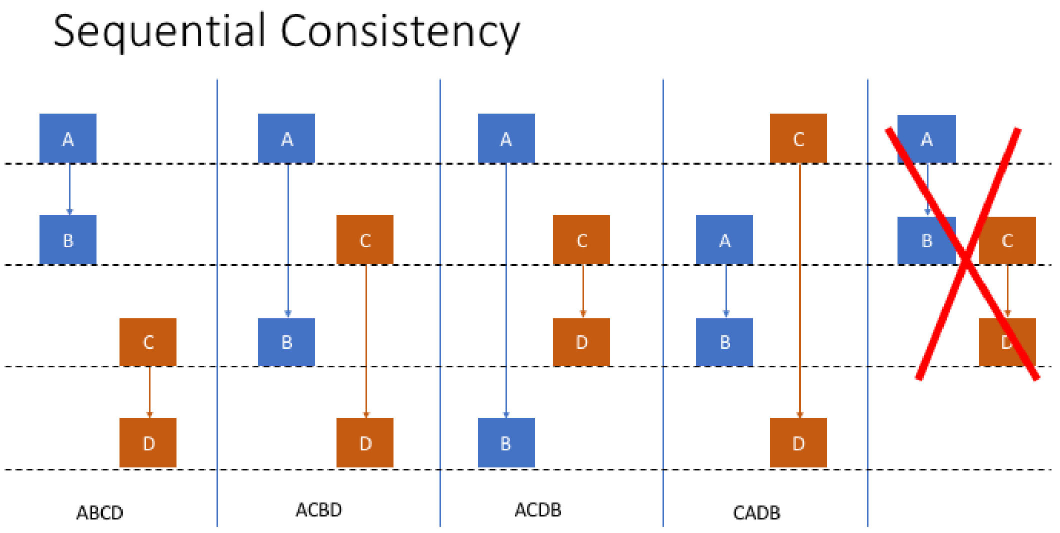


figure 6. shows that what is sequential consistency and what is not

考虑如下程序执行顺序

```
std::atomic_bool x, y;
std::atomic_int z;

void write_x() {
  x.store(true, std::memory_order_seq_cst); //2
}

void write_y() {
  y.store(true, std::memory_order_seq_cst); //3
}

void read_x_then_y() {
  while (!x.load(std::memory_order_seq_cst));
  if (y.load(std::memory_order_seq_cst)) //4
    ++z;
}

void read_y_then_x() {
  while (!y.load(std::memory_order_seq_cst));
  if (x.load(std::memory_order_seq_cst)) //5
    ++z;
}

int main() {
  x = false;
  y = false;
  z = 0;
  std::thread a(write_x);
  std::thread b(write_y);
  std::thread c(read_x_then_y);
  std::thread d(read_y_then_x);
  a.join(); b.join(); c.join(); d.join();
  assert(z.load() != 0); //1, z must be 1 or 2
}
```

code 8. demo of sequential consistency in C++

在多线程情况下，只要不产生 data race 的(data race means undefined behavior), 我们都能为多线程程序找到一个 total order，不管在总体时序上每个指令在各个线程是如何交织(interleaving)的, code 8 中，对于所有的变量(例子中的都是原子变量)，都使用了 `std::memory_order_seq_cst`，意思就是说对于说有相关的变量(包括上下文出现的变量)在编译优化时不能破坏 sequential consistency，同时在生成的 CPU 指令里加入一些额外的同步指令使其在运行时也到保证. 所以无论程序以何种方式运行, code 8 最后一行 `assert` 都会成立，并且各个 线程只看到同一个程序执行顺序的 total order.

如果 assert 失败了(z==0), c, d 两个线程执行结果都是 0 线程 c 执行结果是 0，那么就意味着它看到”store to x happens before store to y”，线程 d 执行结果是 0, 那么就意味着它看到”store to y happens before store to x”，从全局来看是两个完全相反的执行顺序，就是说”store to x”和”store to y”的执行效果的 在各个核的可见性是不统一的，这是一种 data race 的表现.

但是如果把上述的 memory order 换成 `relaxed` 或者 `load acquire` + `store release`，最后的 `assert` 就不一定会成立了(虽然很难复现).

操作原子变量时, relaxed 确保的是 对内存的操作都是原子可见，核内不进行 reorder 限制，同时核之间的同步性能损耗最少，可能只需要在 RMW 时，部分 lock 一下总线即可，不是相同地址的操作还是可 以往总线上读写，这个同步的过程对总线利用比较高.

acquire 或者 release 语义在保证原子性的同时还限制本核内的 reorder 行为，但并不会进行多核同步(同 relaxed)，多核之间可见性的行为限制不了，所以使用这两个语义也达不到顺序一致的效果.

seq_cst 除了保证 relaxed 的原子性以及本核内的 reorder 行为(acqure+release), 同时还确保本核对其他核同步时，使用时总线达到一个”独占”的效果，显然，这个同步的过程时间相对比较长.

load acquire 和 store release 只能保证一个核的序不能保证全局序, relaxed, acquire 以及 release 只能保证原子性，以及本核上对应语义的序，但是不能保证其他核看到本核同步结果的顺序. 而 seq_cst 在同步结果时向其他核进行同步, 确定了一 个全局序.

下图解释了 thread c 和 thread d 两个线程可能看到的 x y 的顺序不一样, a 完成 x=1 之后向其他核同步结果，同时 b 完成 y=1 也向其他核同步结果，但是这个同步的 过程并不确保所有的核在”同一时间”内都看到了自己写的值. c 先看到了 a 的结果，然后看到了 b 的结果，所以认为 x 先写了. d 先看到了 b 的结果，然后看到了 a 的结果，所以认为 y 先写了.

c 和 d 对于序的理解冲突了，要理解这个原因，我们简化模型，考虑 a 在写 x=1 时, b 也在同时写 y=1，而片上总线是同一个，这就和我们并发操作同个数据类似, core3 先看到 x=1 后看到 y=1 core4 先看到 y=1 后看到 x=1

```
thread a        thread b          thread c        thread d
+-----+         +-----+           +-----+         +-----+
|core1|         |core2|           |core3|         |core4|
|     |         |     |          |     |         |     |
+-----+         +-----+           +-----+         +-----+
  x=1             y=1             x==1?           y==1?
   |               |              ^ ^             ^ ^
   |               |              | |             | |
   \________________|_______________/ |             | |
    \              |               |             | |
     \             \_________________|______________/ |
      \             \              |               |
       \______________\_______________|________________/
                     \             |
                      \_____________/
```

如果加上了全局序的限制(seq_cst)，一个可能的序如下，各个核都会先看到 x=1, 然后才看到 y=1，各个核都会认同这个序.

```
thread a        thread b          thread c        thread d
+-----+         +-----+           +-----+         +-----+
|core1|         |core2|           |core3|         |core4|
|     |         |     |          |     |         |     |
+------+        +------+          +------+        +------+
  x=1             y=1             x==1?           y==1?
   |               |              ^ ^             ^ ^
   |               |              | |             | |
   \________________|________________|_|______________/ |
                   |                |               |
                   |                |               |
                   \_________________|_________________/
```

(这个例子几乎对所有支持 SC memory model 的语言都成立，比如 rust)

sequential consistency 在有些时候，可能会有性能瓶颈，因为它需要确保操作在所有线程 之前全局同步，一些操作需要进行排队吞吐就降低了.

C++ 11 默认的 memory model 就是 sequential consistency, 后文会进行详细描述.

为了加深印象，我们再来看 Herb 举的两个描述 SC 更加直观简单例子，不管执行顺序如何，全局所有线程看 到一个相同的顺序(total order).

Transitivity/causality: x and y are std::atomic, all variables initially zero.

```
Thread 1        Thread 2                Thread 3
g = 1;          if(x == 1) y = 1;        if(y == 1) assert(g ==
1);
x = 1;
```

It must be impossible for the assertion to fail – wouldn't be SC.

Total store order: x and y are std::atomic and initially zero.

```
Thread 1      Thread 2      Thread 3                 Thread 4
x = 1;        y = 1;        if (x==1 && y==0)         if (y==1 &&
x==0)
                             print("x first");         print("y
first");
```

It must be impossible to print both messages – wouldn't be SC.

### 5.2 Out of thin air

如果编译器对我们编译出来的代码进行了一些(指令重排的)优化，这些优化并不影响其在单 线程执行的结果，但是如果编译器不遵循一定的 memory model，那么很有可能会编译出在多 线程环境下运行是带有 bug 的汇编代码.

比如说如下例子, out-of-thin-air, compiled with x86-64 gcc 4.1.2 `-O3`.

```
// cpp                      | ;asm code
int v = 0;                  |
                           |
void foo(int set_v) {       | foo(int):
  if (set_v) {              |   mov     %eax, 10086
    v = 10086;              |
test    %edi, %edi             ;test condition
  }                        |   cmove   %eax, DWORD PTR
v(%rip)  ;load maybe
}                          |   mov     DWORD PTR
v(%rip), %eax  ;store always
                           |   ret
```

code 7. out of thin air

可以看到，编译器在优化编译的时候, **为了节省跳转指令**，用了一个 trick，这个 trick 会导致左边源代码的 `if` 条件里边每次都会走到，也就是说即使条件没有满足，每次 `v` 都 会被赋值，这个在单线程里执行是没有问题的，但是如果在多线程环境下, `v` 被别的线程引 用了，并且变更了这个值，事情就变得复杂了，比如说另外一个线程 2 执行如下语句

```
v = 10010;
```

线程 2 对 `v` 的变更可能永远未可见

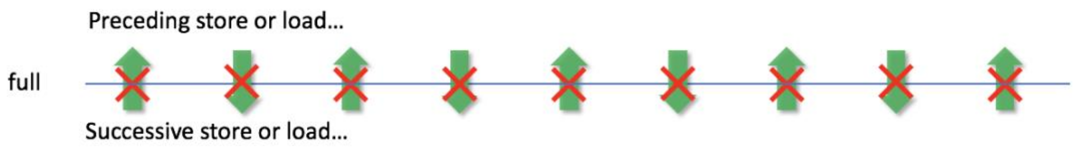
**note: gcc 4.4.7 之后就已经修复了这个问题**

---

`Sequentially Consistent`概念跟`Acquire and Release semantics`是息息相关的，我们知道，程序的编译以及指令的执行在单线程内如果都遵循指令重排的基本原则`Acquire and Release semantics`可以控制单线程内部的顺序，通过配对使用，可以使 线程间构成 synchronizes-with

### 5.3 Full Fence

A full fence is a bidirectional restriction fence which has both acquire and relese semantics.



Nothing goes up and nothing goes down.

Sequential consistency is usually implemented with full fence.

---

## 6 Control instruction reordering

编译器和 CPU 在背地里做了一些"勾当"，目的都是优化，如果我们不知道他们做了什么，很 可能会写出有 bug 的代码，类似这个例子所示，所以在多线程的系统/环境中，我们需要了解并且掌控程序的 reordering.

### 6.1 Low-level reordering constraints

在 code 4 中我们在 c++代码中插入了一条汇编语句

```
    asm volatile("" ::: "memory");
```

这一句其实没有产生任何汇编指令，而是告诉编译器，不要将这一句周围(上下)的指令进行 重排，就是编译期的限制，是一个编译期的`release + acquire semantics`.

如前文所述, code 4 例子中还是能在 runtime 捕获到捕获到指令重排，如果要阻止这种行为，我们需要

```
   asm volatile("mfence" ::: "memory");
```

`mfence` 是一条 Intel 汇编指令，在 Fence Instruction 章节中我们 介绍了几种现代 Intel CPU 的 memory fence 指令，简单再来再提一下

- sfence, store fence，在 instruction stream 中，所有在 sfence 之前的 store 指令都不能 重排到 sfence 之后，但是不影响 sfence 之前的 load 指令重排到 sfence 之后
- lfence, load fence，在 instruction stream 中，所有在 lfence 之前的 load 指令不能重排 到 lfence 之后，但是不影响 lfence 之前的 store 令重排到 lfence 之后
- mfence, “full fence”，在 instruction stream 中，所有在 mfence 之前的 store/load 指令 都不能重排到 mfence 之后

汇编是一个非常底层的语言，也是能精确控制每一步的 ordering，如下代码中，我们可以用 一个 `mfence` 来解决重排的问题，也可以根据实际情况使用

`sfence` or `lfence`

```
// cpp                                  | ;asm
                                        |
extern int a, b;                         |  bar():
                                        |      mov     eax, DWORD
PTR b[rip]
void bar() {                             |      mov     DWORD PTR
a[rip], eax
  a = b;                                |      mfence
  asm volatile("mfence" ::: "memory");  |      mov     DWORD PTR
b[rip], 1
  b = 1;                                |      ret
}                                       |
                                        |
```

code 9. constrains reordering with low-level fence instruction

### 6.2 Lock, Mutex

Locks/Mutexes are a system level mechanism that ensures there is only one thread run at a time, which “perfectly” prevent concurrency and data races.

If we run the program with a single thread, due to the rule of reordering. There won’t be any undefined behaviors introduce by reordering because there is only one thread can stay in `critical section` protected by the lock.

```
// cpp                         | ;asm
std::mutex mtx;                 | bar():
void bar() {                    |   push    rbx
  {                            |   mov     ebx, OFFSET
FLAT:_ZL28__gthrw___pth...
    lock_guard<mutex> l(mtx); |   test    rbx, rbx
    a = b;                     |   je      .L9
  }                            |   mov     edi, OFFSET FLAT:mtx
  b = 1;                       |   call
__gthrw_pthread_mutex_lock(pthread_mutex_t*)
}                              |   test    eax, eax
                               |   jne     .L21
                               | .L9:
                               |   mov     eax, DWORD PTR b[rip]
                               |   mov     DWORD PTR a[rip],
eax ;;;assign a;;;;
                               |   test    rbx, rbx
                               |   je      .L10
                               |   mov     edi, OFFSET FLAT:mtx
                               |   call
__gthrw_pthread_mutex_unlock(pthread_mutex_t*)
                               | .L10:
                               |   mov     DWORD PTR b[rip],
1   ;;;assign b;;;;
                               |   pop     rbx
                               |   ret
                               | .L21:
                               |   mov     edi, eax
                               |   call
std::__throw_system_error(int)
```

code 10. prevent reordering with lock, compiled with `x86-64 gcc 8.2 -O3`

BTW, a lock carries the acquire and release semantics, usually both of them, the underlying implementation of locks/mutexes usually introduces low-level full memory fences, and also work as a full fence at runtime – nothing can pass the lock.

Briefly speaking, lock – acquire semantics, unlock – release semantics.

```
  std::unique_lock<std::mutex> lock(mtx);

  lock.lock();  // nothing following in the critical section
reordered
               // above this lock(), acquire semantics

  // critical section
  ...

  lock.unlock(); // nothing prior in the critical section
reordered
                // past this unlock(), release semantics
```

### 6.3 Memory Fences

Memory Fence 和 Memory Barrier 其实是同个意思，中文翻译应该都叫内存屏障(叫内存栅栏好像有点不那么顺口:smirk:)，只是英文叫法不一样，看多了 Linux 内核的人可能更加习惯叫 barrier，但是我更习惯叫 Memory Fence，因为相关的 汇编指令就叫 fence.

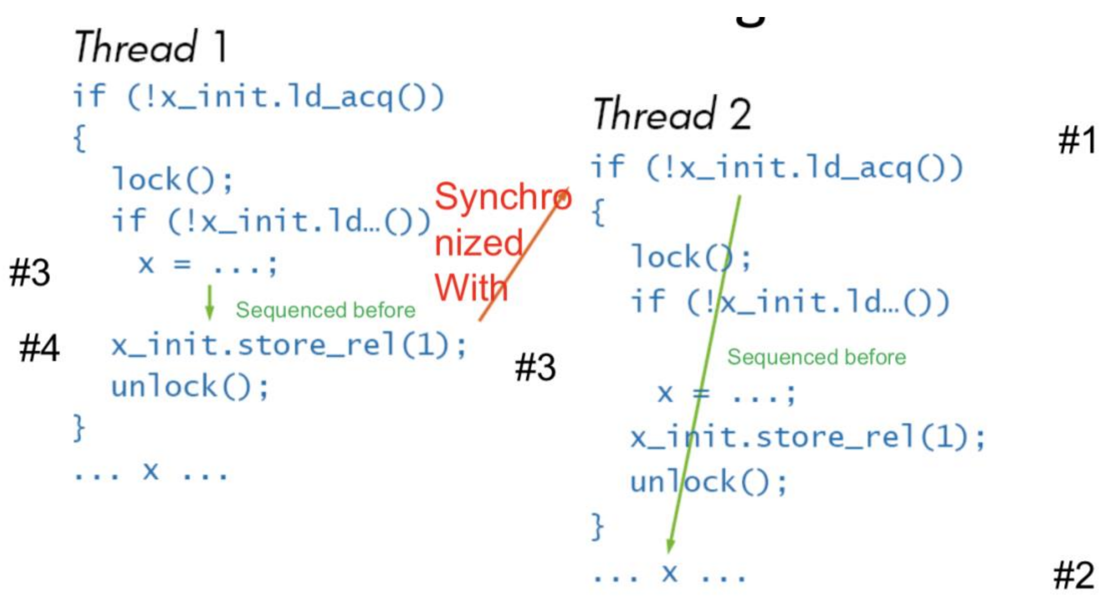
在之前的 code 4 和 code 2b，我们已经展示了从**编程语言层面** 如何通过添加语句 `asm volatile("" ::: "memory")`(汇编，关于 gcc `asm` 使用的说明在 这里) 或者 `std::atomice_thread_fence`(C++11)来限制编译期(乃至运行期)的 reordering.

语言层面对于 memory fence 这个概念的抽象和支持，将会在接下来的章节 详细阐述，一个好的抽象可以非常好地实现 memory fence 并且在各种场景下得到很好的优化 .

# 7 Synchronizes-with relation

The following concepts are used to describe sequence of each operation in multi-thread environment, we just need to know that:

- synchronizes-with inter thread, release and acquire semantics description
- sequenced-before intra thread order description
- happens-before inter thread order description



```
Thread 1
if (!x_init.ld_acq())
{
  lock();
  if (!x_init.ld…())
    x = ...;
  x_init.store_rel(1);
  unlock();
}
... x ...
```

```
Thread 2
if (!x_init.ld_acq())
{
  lock();
  if (!x_init.ld…())
    x = ...;
  x_init.store_rel(1);
  unlock();
}
... x ...
```

Michael Wong's slides shows the idea of sequenced-before + memory-ordering to implement happens-before and synchronizes-with.

# 8 Memory Model of C++11

What is software memory model:

- A set of rules that describe allowable semantics for memory accesses on a computer program
  - Defines the expected value or values returned by any loads in the program
  - Determines when a program produces undefined behavior (e.g load from uninitialized variables)
- Stroustrup, The C++ Programming Language 4th Edition §41.2 represents a contract between the implementers and the programmers to ensure that most programmers do not have to think about the details of modern computer hardware
- Critical component of concurrent programming

The default memory model is sequentially consistent, and the memory order is a very important, may be the most important, part of C++ memory model.

## 8.1 Memory Order of C++11

The C++ standard says that (namespace `std` is omitted here):

- `memory_order_relaxed`: allow reordering, no explicit fence

  Relaxed operation: there are no synchronization or ordering constraints imposed on other reads or writes, only this operation’s atomicity is guaranteed.

- `memory_order_consume`: keep data dependency only, reordering may still

  happen, weaker than `acquire` and can be replaced entirely with `acquire`.

  It’s discouraged to use `consume`.

  A load operation with this memory order performs a consume operation on the affected memory location: no reads or writes in the current thread dependent on the value currently loaded can be reordered before this load. Writes to data-dependent variables in other threads that release the same atomic variable are visible in the current thread. On most platforms, this affects compiler optimizations only.

- `memory_order_acquire`: any memory access cannot be reordered

  upwards this point, any associated atomic variables can be accessed correctly

  A load operation with this memory order performs the acquire operation on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread.

- `memory_order_release`: any memory access cannot be reordered

  downwards this point, all memory precedes this point is visible to other threads

  A store operation with this memory order performs the release operation: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable (see Release-Acquire ordering below) and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic.

- `memory_order_acq_rel`: `acquire` + `release`, works for RMW

  operation such as `fetch_add`

  A read-modify-write(RMW) operation with this memory order is both an acquire operation and a release operation. No memory reads or writes in the current thread can be reordered before or after this store. All writes in other threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable.

- `memory_order_seq_cst`: load acquire + store release + single total order

  A load operation with this memory order performs an acquire operation, a store performs a release operation, and read-modify-write performs both an acquire operation and a release operation, plus a single total order exists in which all threads observe all modifications in the same order.

The definition of each memory ordering is very clear, they intend to fulfill the C++ memory model, and they can be applied to the function families as following,

see `<atomic>` for more info.

```
std::atomic<T>::load()
std::atomic<T>::store()
std::atomic<T>::compare_and_exchange_*()
std::atomic<T>::exchange()
std::atomic_thread_fence() // standalone fence
std::atomic_load()
std::atomic_store()
std::atomic_compare_and_exchange()
std::atomic_exchange()
```

(from here on in this section, namespace `std` may be omitted in code snippet)

We can conclude the strength of constraints of the memory ordering as follows:

```
                  /----------> release ----------\
                 /                                \
      relaxed --->                                 ---> acq_res --
-> seq_cst
                 \                                /
                  \---> consume ---> acquire ----/

an arrow A -> B means constraint of B is stronger than A
```

**And we should note that, by definition in the c++ standard of [acquire and release semantics](), `memory_order_consume` and `memory_order_acquire` will not make sense for atomic load operations; `memory_order_release` will not make sense for atomic store operations.**

```
atomic<int> A;

A.store(10086, memory_order_acquire); // does not make sense,
may be undefined behavior
A.store(10086, memory_order_consume); // does not make sense,
may be undefined behavior
A.load(10086, memory_order_release); // does not make sense,
may be undefined behavior

A.store(10086, memory_order_relaxed); // store-relaxed
operation
A.store(10086, memory_order_release); // store-release
operation
A.store(10086, memory_order_acq_rel); // store-release
operation
A.store(10086, memory_order_seq_cst); // SC store-release
operation

A.load(10086, memory_order_relaxed); // load-relaxed operation
A.load(10086, memory_order_consume); // load-consume operation
A.load(10086, memory_order_acquire); // load-acquire operation
A.load(10086, memory_order_acq_rel); // load-acquire operation
A.load(10086, memory_order_seq_cst); // SC load-acquire
operation
```

```
A.fetch_add(10086, memory_order_relaxed); // store-relaxed
operation
A.fetch_add(10086, memory_order_release); // store-release
operation
A.fetch_add(10086, memory_order_acq_rel); // load-acquire +
store-release operation
A.fetch_add(10086, memory_order_seq_cst); // SC load-acquire +
store-release operation
```

`memory_order_consume` may be a little confusing, this memory order looses the load-acquire constraint by considering the data dependency, to explain it in detail I illustrate a code snippet here.

```
extern int X;
void foo() {
  atomic<int> A {10086};
  int B = 10010;
  int C = 10000;

  X = A.load(memory_order_consume); // a "consume fence"

  B = B + 1; // may/can be reordered before x
  C = A.load(memory_order_relaxed) + 1; // can not be reordered
before X
                                        // because it depends on A
}
```

Note that every `consume` can be replaced with `acquire`, but not vice versa, the data dependency is some time hard to figure out, and don't be clever when you don't fully understand the consequences by replacing `acquire` with `consume` if you think it "may" improve the performance.

`consume` in most cases is a compiler option, the code gen is almost the same as `acquire`, `consume` is a compiler optimization option only.

---

### 8.2 Standalone fence, stronger but slower

Standalone fence is described in section 29.8 of working draft.

*Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an acquire fence. A fence with release semantics is called a release fence.*

A fence can be generated with `std::atomic_thread_fence()` and corresponding `std::memory_order`.

A fence with acquire semantics is called an acquire fence, `std::atomic_thread_fence(std::memory_order_acquire)`. A fence with release semantics is called a release fence, `std::atomic_thread_fence(std::memory_order_release)`.

Let’s analyse a code snippet:

```
extern int tmp;

tmp = 10086;
A.store(10010, std::memory_order_release);
```

is it equivalent to the following?

```
extern int tmp;

tmp = 10086;
std::atomic_thread_fence(std::memory_order_release); //
release fence
// can be reorded? float up?
A.store(10010, std::memory_order_relaxed);
```

There may be a mis-concept that store to A with `memory_order_relaxed` can be reordered upwards that standalone release fence, due to mis-understanding of release operation

*Release semantics is a property that can only apply to operations that write to shared memory, whether they are read-modify-write operations or plain stores. The operation is then considered a write-release. Release semantics prevent memory reordering of the write-release with any read or write operation that precedes it in program order.*

Release semantics constrains nothing will be reordered after the write-release, but cannot prevent some other operations from floating past the write-release, and even past that store `tmp = 10086` before release fence, if standalone release fence performs in the same way, it won't make any sense that we still need a standalone fence.

Fortunately, C++ standard disallows that reordering before release fence, its the same for the acquire fence too.

N3337 29.8.2 says:

*A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.*

It may be a little hard to understand, to explain this, I draw this figure:

```
                            sync     Y          M.load
A   ===^===^===^===^===^===   ------>   B
===v===v===v===v===v===
X         M.store
```

The constraints/consequences are:

1. A synchronizes with B
2. X cannot float past A — X cannot be reordered. X is allowed to be reordered before A if it is pure release semantics
3. Y cannot sink past B — Y cannot be reordered. Y is allowed to be reordered past B if it is pure acquire semantics

In other words by cppreference:

`atomic_thread_fence` imposes stronger synchronization constraints than an atomic store operation with the same `std::memory_order`. While an atomic store-release operation prevents all preceding writes from moving past the store-release, an `atomic_thread_fence` with `memory_order_release` ordering prevents all preceding **writes** from moving past **all subsequent stores**.

**Standalone `atomic_thread_fence` with `memory_order_release` or `memory_order_acquire` is stricter than store-release or load-acquire operation.**

This post by Jeff Preshing discusses the example given previously in details.

And what we need to notice is that standalone fences are sub-optimal and it is a performance pessimization.

---

## 8.3 `compare_exchange_weak` or `compare_exchange_strong`?

Sometimes it maybe confusing that which one to use for CAS, `compare_exchange_weak` or `compare_exchange_strong`, for cpp users, as always we should analyse the difference and than choose which on to use.

Here is my conclusion on this question:

- `compare_exchange_weak` may fail with spurious error, even if the expected value matched due to some reasons (system level or hardware level)
- `compare_exchange_weak` is "cheaper" than `compare_exchange_strong`, as if the strong one keeps trying in a loop to exchange when "spurious error" occurs with the weak one
- conventionally, which to use:
    - if there is a CAS loop (spin), use `compare_exchange_weak`

```
o   while (compare_exchange_weak(:::)) { ::: }
```

- if there is a single test, use `compare_exchange_strong`

```
o   if (compare_exchange_strong(:::)) { ::: }
```

`compare_exchange_strong/weak` is conditional read-write-modify operation, also called compare-and-swap (CAS).

Naive CAS implementation on hardware may look like this (pseudo code):

```
bool compare_exchange_strong(T& old_v, T new_v) {
 Lock L;       // Get exclusive access
 T tmp = value; // Current value of the atomic
 if (tmp != old_v) { old_v = tmp; return false; }
 value = new_v;
 return true;
}

// Lock is not a real mutex but some form of exclusive access
implemented in hardware
```

Notice that read is faster than write, we can read first and than lock, the implementation can be faster:

```
bool compare_exchange_strong(T& old_v, T new_v) {
 T tmp = value;                               // Current value
of the atomic
 if (tmp != old_v) { old_v = tmp; return false; }
 Lock L;                                     // Get exclusive
access
 tmp = value;                                 // value could
have changed!
 if (tmp != olv_v) { old_v = tmp; return false; }
 value = new_v;
 return true;
}
```

```
// Double-checked locking pattern is back!
```

If exclusive access is hard(expensive) to get, use weak lock other than strong lock, let someone else try:

```
bool compare_exchange_weak(T& old_v, T new_v) {
 T tmp = value;                                // Current value
of the atomic
 if (tmp != old_v) { old_v = tmp; return false; }
 TimedLock L;                                  // Get exclusive
access or fail
 if (!L.locked()) return false;                  // old_v is
correct
 tmp = value;                                  // value could
have changed!
 if (tmp != olv_v) { old_v = tmp; return false; }
 value = new_v;
 return true;
}
```

We can see why there is **spurious error** for `compare_exchange_weak` by explaining with pseudo code: CPU reads the value and passes the check but it fails to “lock” the “bus”, the call of `compare_exchange_weak` returns false.

From the pseudo code implementations, `compare_exchange_weak` and `compare_exchange_strong` seems to be the same expensive, why do we say that `compare_exchange_weak` is cheaper/faster? When `TimedLock` fails, there is another core succeeds, if `TimedLock` is much easier or more light-weight than `Lock` to make progress, the **overall** performance will be better with `compare_exchange_weak`.

`compare_exchange_weak` and `compare_exchange_strong` act the same on x86-64, however, on SPARC or ARM, they are not the same. Performance varies from platform to platform.

## 9 volatile

`volatile` in C++ is only a compile-time keyword which prevents the compiler reordering the specified (such as redundant read), that a hint only, but no atomicity at runtime guaranteed.

A typical case

```
extern int a;
volatile int b = a; // load a
// do somthing that not related to a
...
b = a; // load a again, this redundant statement won't be
reordered or optimized
```

`volatile` in Java or C#, it kind of means `atomic`, the write or read to the specified var is "atomic", no partial write or read will happen.

And also, note that

- in .NET and Java key word `volatile` ensures sequentially consistent memory order, kind of like `atomic` with default memory order in C++
- C++ `std::atomic` default memory order is sequential-consistency we can weaken that, but I didn't find out how to do that with `volatile` in .NET or Java.

## 10 Code generation

This section introduces the code generation that implements C++ memory model on different types of processors.

They are compiled with gcc 8.2, 8.2, 6.3, 5.4 with option `-std=c++17 -O3`. The versions of gcc varies because they are the newest gcc version for corresponding processors on .

```
int a = 1;
int b = 2;
std::atomic<int> a1 {3};
extern int c;

void foo() {
  a = 1;
  b = 2;
  a1.compare_exchange_strong(a, 10086,
std::memory_order_relaxed);
  a = c;
}
```

code. code snippet for code generation

| operation | x86-64 | power64le |
|---|---|---|
| atomic_thread_fence(seq_cst) | mfence | sync;blr |
| atomic_thread_fence(acq_rel) | - | lwsync;blr |
| atomic_thread_fence(release) | - | lwsync;blr |
| atomic_thread_fence(acquire) | - | lwsync;blr |
| atomic_thread_fence(consume) | - | lwsync;blr |
| atomic.store(seq_cst) | mov;mfence | li;addis;sync;stw |
| atomic.store(acq_rel) | mov;mfence | li;addis;sync;stw |
| atomic.store(release) | mov | li;addis;sync;stw |
| atomic.store(relaxed) | mov | li;stw |
| atomic.load(seq_cst) | mov | sync;addis;lwz,cmpw,bne*;isync |
| atomic.load(acq_rel) | mov | sync;addis;lwz,cmpw,bne*;isync |
| atomic.load(acquire) | mov | sync;addis;lwz,cmpw,bne*;isync |
| atomic.load(relaxed) | mov | addis;lwz |
| atomic.exchange(seq_cst) | xchg | sync;lwarx;stwcx;bne*;isync |
| atomic.exchange(acq_rel) | xchg | lwsync;lwarx;stwcx;bne*;isync |
| atomic.exchange(acquire) | xchg | lwarx;stwcx;bne*;isync |
| atomic.exchange(consume) | xchg | lwarx;stwcx;bne*;isync |
| atomic.exchange(release) | xchg | lwsync;lwarx;stwcx;bne* |

| operation | x86-64 | power64le |
|---|---|---|
| atomic.exchange(relaxed) | xchg | lwarx;stwcx;bne* |
| atomic.compare_exchange_strong(seq_cst) | lock cmpxchg | sync;lwarx;cmpwi;bne*;stwcx.;bne*; |
| atomic.compare_exchange_strong(acq_rel) | lock cmpxchg | lwsync;lwarx;cmpwi;bne*;stwcx.;bne |
| atomic.compare_exchange_strong(acquire) | lock cmpxchg | lwarx;cmpwi;bne*;stwcx.;bne*;isync |
| atomic.compare_exchange_strong(consume) | lock cmpxchg | lwarx;cmpwi;bne*;stwcx.;bne*;isync |
| atomic.compare_exchange_strong(release) | lock cmpxchg | lwsync;lwarx;cmpwi;bne*;stwcx.;bne |
| atomic.compare_exchange_strong(relaxed) | lock cmpxchg | lwarx;cmpwi;bne*;stwcx.;bne* |
| atomic.fetch_add(seq_cst) | lock add/xadd | sync;lwarx;stwcx.;bne*;isync |
| atomic.fetch_add(acq_rel) | lock add/xadd | lwsync;lwarx;stwcx.;bne*;isync |
| atomic.fetch_add(acquire) | lock add/xadd | addis;addi;lwarx;addi;stwcx.;bne*; |
| atomic.fetch_add(consume) | lock add/xadd | addis;addi;lwarx;addi;stwcx.;bne*; |
| atomic.fetch_add(release) | lock add/xadd | lwsync;lwarx;stwcx.;bne* |
| atomic.fetch_add(relaxed) | lock add/xadd | addis;addi;lwarx;addi;stwcx.;bne* |

table. code generation part 1

the `fetch_add/fetch_sub` will generate `add` or `xadd` on x86-64 according to

the use of reuturn value, if not used simple `lock add` is enough.

| operation | arm64 | mips64 |
|---|---|---|
| atomic_thread_fence(seq_cst) | dmb | sync |
| atomic_thread_fence(acq_rel) | dmb | sync |
| atomic_thread_fence(release) | dmb | sync |
| atomic_thread_fence(acquire) | dmb | sync |
| atomic_thread_fence(consume) | dmb | sync |

| operation | arm64 | mips64 |
|---|---|---|
| atomic.store(seq_cst) | stlr | li;sync |
| atomic.store(acq_rel) | stlr | li;sync |
| atomic.store(release) | stlr | li;sync |
| atomic.store(relaxed) | str | li |
| atomic.load(seq_cst) | ldar | ld;sync;lw |
| atomic.load(acq_rel) | ldar | ld;sync;lw |
| atomic.load(acquire) | ldar | ld;sync;lw |
| atomic.load(relaxed) | ldr | ld;lw |
| atomic.exchange(seq_cst) | ldaxr;staxr;cbnz* | sync;li;sc;beq*;syn |
| atomic.exchange(acq_rel) | ldaxr;staxr;cbnz* | sync;li;sc;beq*;syn |
| atomic.exchange(acquire) | ldaxr;stxr;cbnz* | li;sc;beq*;sync |
| atomic.exchange(consume) | ldaxr;stxr;cbnz* | li;sc;beq*;sync |
| atomic.exchange(release) | ldxr;staxr;cbnz* | sync;li;sc;beq* |
| atomic.exchange(relaxed) | ldxr;stxr;cbnz* | li;sc;beq* |
| atomic.compare_exchange_strong(seq_cst) | ldaxr;cmp;bne;stlxr;cbnz* | sync;ll;bne*;li;sc; |
| atomic.compare_exchange_strong(acq_rel) | ldaxr;cmp;bne;stlxr;cbnz* | sync;ll;bne*;li;sc; |
| atomic.compare_exchange_strong(acquire) | ldaxr;cmp;bne;stxr;cbnz* | ll;bne*;li;sc;beq*; |
| atomic.compare_exchange_strong(consume) | ldaxr;cmp;bne;stxr;cbnz* | ll;bne*;li;sc;beq*; |
| atomic.compare_exchange_strong(release) | ldxr;cmp;bne;stlxr;cbnz* | sync;ll;bne*;li;sc; |
| atomic.compare_exchange_strong(relaxed) | ldxr;cmp;bne;stxr;cbnz* | ll;bne*;li;sc;beq*; |
| atomic.fetch_add(seq_cst) | ldaxr;add;stlxr;cbnz* | sync;ll;addiu;sc;be |
| atomic.fetch_add(acq_rel) | ldaxr;add;stlxr;cbnz* | sync;ll;addiu;sc;be |
| atomic.fetch_add(acquire) | ldaxr;add;stxr;cbnz* | sync;ll;addiu;sc;be |
| atomic.fetch_add(consume) | ldaxr;add;stxr;cbnz* | sync;ll;addiu;sc;be |
| atomic.fetch_add(release) | ldxr;add;stlxr;cbnz* | sync;ll;addiu;sc;be |
| atomic.fetch_add(relaxed) | ldxr;add;stxr;cbnz* | sync;ll;addiu;sc;be |

table. code generation part 2

`*` means there is a loop

x86-64's code generation is much simpler compared to the rest due to it's strong hardware memory model, but there is no free lunch, the `lock` instruction locks the whole bus, it's very expensive.

With stronger constraints, comes less flexibility for optimization.

There is no free lunch for ARM, POWER and MIPS too. Although they are weaker than x86-64 and more flexible, however, it's easier to write buggy code while doing optimization.

---

# 11 Lock-free programming

Definition, examples of lock-free programming, and usage of atomics and memory ordering for lock-free.

Lock-free programming is doing trick with CAS at most of time, we need to consider all of the corner cases under concurrent situations.

In my point of view, **it's the key factor to ensure your lock-free program is correct iff what you operate is what you want to operate, even if it keeps changing all the time**

## 11.1 Definition - what is lock-free

*The Lock-Free property guarantees that at least some thread is doing progress on its work. In theory this means that a method* ***may take an infinite amount*** *of operations to complete, but in practice it takes a short amount, otherwise it won't be much useful.*

*Definition: A method is Lock-Free if it guarantees that infinitely often some thread calling this method finishes in a finite number of steps.*

*The Wait-Free property guarantees that any given thread provided with a time-slice will be able to make some progress and eventually complete. It guarantees that* ***the number of steps is finite****, but in practice it may be extremely large and depend on the number of active threads, which is why there aren't many practical Wait-Free data structures.*

*Definition: A method is Wait-Free if it guarantees that every call finishes its execution in a finite number of steps.*

The above is quoted from this article, which is explained very well.

To understand it in a simple way, I put strengths of lock-freedom and properties as following:

- Not lock-free

    - explicit lock/mutex
- Lock-free, “someone makes progress”
    - no explicit lock/mutex
    - every step taken achieves global progress (for some sensible definition of progress)
    - guarantee system-wide throughput
- Wait-free, “no one ever waits”
    - every operation will complete in a bounded #steps no matter what else is going on
    - guarantee system-wide throughput and starvation-freedom
    - all wait-free algorithms are lock-free, but not vice versa

When we say that a particular data structure is Lock-Free it means that all its operations are Lock-Free, or better.

We should analyse the program to see what kind of “lock-free” it is, program without explicit locks is lock-free but performance varies among different implementations.

### 11.1.1 Lock-freedom examples

This is a piece of code that is lock-free but not wait-free.

```
size_t sleep_micro_sec = 5;
std::atomic<bool> locked_flag_ = ATOMIC_VAR_INIT(false);

void foo() {
  bool exp = false;
  while (!locked_flag_.compare_exchange_strong(exp, true)) {
    exp = false;
    if (sleep_micro_sec == 0) {
      std::this_thread::yield();
    } else if (sleep_micro_sec != 0) {

std::this_thread::sleep_for(std::chrono::microseconds(sleep_mi
cro_sec));
    }
  }
}
```

This is a piece of code that is (usually) wait-free with low(not that high) contention. We said it is “usually” because it may spin for long with high contention.

```
template<typename T>
void slist<T>::push_front(const T& t) {
  auto p = new Node;
  p->t = t;
  p->next = head;
  // spin until success
  // try to prepend to head when we see the head
  while (!head.compare_exchange_weak(p->next, p)) { }
}
```

## 11.2 Double-checked lock pattern, DCLP

GOF’s singleton pattern, lazy evaluation

```
Singleton* Singleton::instance = nullptr;

Singleton* Singleton::get_instance() {
  if (instance == nullptr) {
    lock_guard<mutex> lock(mtx);
    if (instance == nullptr) {
      instance = new Singleton(); // reordering or visibility
of instance issue
    }
  }
  return instance;
}
```

This may not work, due to other threads may see partially constructed instance, if Singleton() is inlined explicitly or optimized by compiler.

The writes that initialize the Singleton object and the write to the instance field can be done or perceived out of order.

Even if the compiler does not reorder those writes, on a multiprocessor the processor or the memory system may reorder those writes, as perceived by a thread running on another processor.

To rescue that, use atomic for variable instance:

```
atomic<Singleton*> Singleton::instance {nullptr};
```

It’s sequential consistency by default, all write in Singleton() will and must be done before assigning to instance, no reordering to the assignment.

There are also other approach, like Scott Mayer’s, easy and simple

```
Singleton* Singleton::get_instance() {
  static Singleton a;
  return &a;
}
```

Here’s another variant, a lock free approach with an extra flag.

```
atomic<Singleton*> Singleton::instance {nullptr};
atomic<bool> Singleton::create {false};

Singleton* Singleton::get_instance() {
  if (instance.load() == nullptr) {
    if (!create.exchange(true)) // try to construct
      instance = new Singleton(); // construct
    else while(instance.load() == nullptr) { } // spin
  }
  return instance.load();
}
```

the above code use default memory order – sequential consistency, too strong for this snippet.

```
atomic<Singleton*> Singleton::instance {nullptr};
atomic<bool> Singleton::create {false};

Singleton* Singleton::get_instance() {
  if (instance.load(memory_order_acquire) == nullptr) {
    if (!create.exchange(true, memory_order_acq_rel)) // strong
enough?
      instance.store(new Singleton(), memory_order_release); //
construct
    else while(instance.load(memory_order_acquire) == nullptr)
{ } // spin
```

```
  }
  return instance.load(memory_order_acquire);
}
```

### 11.2.1 Lazy evaluation

The right way for lazy evaluation, using `std::call_once`

```
Singleton* Singleton::get_instance() {
  static std::once_flag create;
  std::call_once(create, [] { instance = new Singleton(); });
  return instance;
}
```

---

## 11.3 Lock-free queue

“Brute-force” lock-free singly-linked list, no pop.

Fixed size vector with atomic cursor and semaphore, one-to-many producer-consumer pattern.

### 11.3.1 Lock-free singly-linked list

```
template <typename T>
class slist {
public:
  slist();
  ~slist();
  void push_front(const T& t);
  void pop_front();
private:
  struct Node {
    T t;
    Node* next;
  }
  atomic<Node*> head {nulllptr};
};
```

Constructor doesn’t need to do anything, but destructor need to recycle internal nodes.

Destructor

```cpp
template<typename T>
slist<T>::~slist() {
  auto first = head.load();
  while (first != nullptr) {
    auto unliked = first;
    first = first-next;
    delete unliked;
  }
}
```

`push_front()` is easy with atomic weapons, to deal with concurrent insertions, we just need to prepend to the head we see exactly with CAS.

```cpp
template<typename T>
void slist<T>::push_front(const T& t) {
  auto p = new Node;
  p->t = t;
  p->next = head;
  // spin until success
  // try to prepend to head when we see the head
  while (!head.compare_exchange_weak(p->next, p)) { }
}
```

`pop_front()` is that easy too?

```cpp
template<typename T>
void slist<T>::pop_front() {
  auto p = head.load(); // load once
  // spin until success
  // move head pointer "backwards" if we see the head
  while (p != nullptr
        && !head.compare_exchange_weak(p, p->next)) { }
  // problem:
```

```
  // ABA and dereferrencing deleted p by p->next in other
threads after deletion
  delete p;
}
```

It seems that works fine, but there are problems:

1. ABA problem
2. Deference deleted pointer

case for ABA problem:

```
Initial state:
  +-------+    +--------+    +--------+    +--------+    +--------+
  | head o--->| addr5 o--->| addr4 o--->| addr3 o--->| addr2 o-
-->
  +-------+    +--------+    +--------+    +--------+    +--------+

Modify the list:

   |     Thread 1                Thread 2
Thread3
   |     // pop                  // pop                        // push
   t
   i     head = 5
   m t1 -------------------------------------------------------
------
   e                            head = 5
   |                            CAS(5, 4)
   | t2 -------------------------------------------------------
------
   |                                                        head = 4
   l                                                        CAS(4,
9)
   i                                                        head = 9
   n                                                        CAS(9,
5)
   e t2 -------------------------------------------------------
------
   |     CAS(5, 4)
   | t4 -------------------------------------------------------
------
   v
```

```
t4:
  +-------+    +=======+    +=======+    +-------+    +-------+
+-------+
  | head o    | addr5'o--->| addr9 o--->| addr4 o--->| addr3 o-
-->| addr2 o--->
  +---o--+    +=======+    +=======+    +-------+    +-------+
+-------+
       \                                   ^
        \_________________________________/
```

case for dereferencing deleted pointer problem (thread 2 dereference a mangling pointer deleted by thread 1):

```
// for convinence of explaination, pop_front() also can be
written as
template<typename T>
void slist<T>::pop_front() {
  auto p = head.load();
  do {
    p = head.load();
    auto n = p->next; // dereference p, which is mean to be
head
  } while (p != nullptr && !head.compare_exchange_weak(p, n));
  delete p;
}

// think of 2 concurrent pops, vertical direction shows in
which time elapses

//  thread 1                                 |  thread 2
                                            |
void slist<T>::pop_front() {                  | void
slist<T>::pop_front() {
  auto p = head.load();                      |   auto p =
head.load();
  do {                                      |   do {
    p = head.load();                         |
    auto n = p->next;                        |     p = head.load();
  } while (p != nullptr                      |
    && !head.compare_exchange_weak(p, n)); |
  delete p;                                 |
}                                                   |      auto n = p->next;
// BANG!
                                                    |    } while (p !=
nullptr
                                                    |
&& !head.compare_exchange_weak(p, n));
                                                    |    delete p;
                                                    v  }
```

There are some (common and may be more) hints to solve the ABA problem:

*We need to solve the ABA issue: Two nodes with the same address, but different identities (existing at different times).*

1. *Option 1: Use lazy garbage collection.*
    - *Solves the problem. Memory can't be reused while pointers to it exist.*
    - *But: Not an option (yet) in portable C++ code, and destruction of nodes becomes nondeterministic.*
2. *Option 2: Use reference counting (garbage collection).*
    - *Solves the problem in cases without cycles. Again, avoids memory reuse.*
3. *Option 3: Make each pointer unique by appending a serial number, and increment the serial number each time it's set.*
    - *This way we can always distinguish between A and A'.*
    - *But: Requires an atomic compare-and-swap on a value that's larger than the size of a pointer. Not available on all hardware & bit-nesses.*
4. *Option 4: Use hazard pointers.*
    - *Maged Michael and Andrei Alexandrescu have covered this in detail.*
    - *But: It's very intricate. Tread with caution.*
5. *...*

It's getting complicated and hard, and of course, if we don't recycle memory there won't be any ABA problem at all! :smirk: Think about it and don't recycle anything if you push a lot and pop is much less, it will work well.

There is an attempt for option 1, 2 and 4 https://stidio.github.io/2017/01/cpp11_atomic_and_lockfree_program/, it may be a bit more complicated, I will discus it in future post (lock-free application series?), if I had time and found that this topic is still interesting.

For option 3, here is an attempt, apparently this attempt has hardware limitation. The idea is using a separate head structure with version.

Comparing the content of the head instead of pure address which may be changed without notifications, which is the essence of solving the ABA problem.

```cpp
template <typename T>
class slist {
...
  struct Node {
    T t;
    Node* next;
  }
  struct HeadNode { // 128 bit
    size_t ver = 0;
    Node* ptr = nullptr;
  }
  // this may be an issue
  atomic<HeadNode> head {0, nulllptr};
};

template<typename T>
void slist<T>::push_front(const T& t) {
  auto p = new Node;
  auto h = head.load();
  do {
    p->next = h.ptr;
    p->t = t;
    HeadNode new_head {h.ver + 1, p}; // increase ver
  } while (!head.compare_exchange_weak(h, new_head));
}

template<typename T>
void slist<T>::pop(const T& t) {
  auto h = head.load();
  do { // does not dereference the pointer to data
    HeadNode new_head {h.ver + 1, h.ptr->next}; // increase ver
  } while (!head.compare_exchange_weak(h, new_head));
  delete h.ptr;
}
```

It resolves the problems mentioned before:

- every time the head has been updated, version of head increased, ABA problem resolved
- we don't dereference any pointers in the linked list, porblem of dereferencing deleted pointer resolved

However, it's not perfect, it may not work as lock-free on all hardwares due to the "big" structure `NodeHead`, some hardware may not have large enough register, here may be 128 bits, to complete CAS within one instruction. To check if it works, the simplest way may be using `std::atomic_is_lock_free()` to determine it.

The following show that atomicity is hardware dependent. (`__atomic_load`)

`gcc8.2 -std=c++17 -O3`

```
// cpp                                  | ; asm
struct alignas(16) BigStruct16 {          |
test_big_struct_lock_free1():
  size_t ver;                            |        sub     rsp, 24
  int64_t data;                          |        mov     esi, 5
};                                      |        mov     rdi, rsp
                                        |        call
__atomic_load_16
struct alignas(64) BigStruct64 {          |        add     rsp, 24
  size_t ver;                            |        ret
  int64_t data;                          |
};                                      |
test_big_struct_lock_free2():
                                        |        push    rbp
void test_big_struct_lock_free1() {   |        mov     ecx, 5
  std::atomic<BigStruct64> a64;        |        mov     edi, 32
  auto c = a64.load();                 |        mov     rbp, rsp
}                                       |        and     rsp, -32
                                        |        sub     rsp, 64
void test_big_struct_lock_free2() {   |        lea     rdx,
[rsp+32]
  std::atomic<BigStruct64> a64;        |        mov     rsi, rsp
  auto c = a64.load();                 |        call
__atomic_load
}                                       |        leave
                                        |        ret
```

```
// on most of modern Intel's x86-64 processors there are 128-
bit registers
// BigStruct16 is lock free
```

#### 11.3.1.1 Conclusion

To make this list pop single element is not that easy, to get rid of ABA, we may also first pop all the elements at once?

```
template<typename T>
auto slist<T>::pop_all() {
  return head.exchange(nullptr);
}
```

Nice, we are finally done.

(Wait, we didn't talk about which memory order to be apply to atomic operations yet?)

---

## 11.4 shared_ptr ref_count

Bug mentioned by Herb in his talk atomic weapons.

It is very interesting and helpful for us to understand memory order much better.

Fist, we give the correct (fixed) code for shared ptr's reference counting.

Increment

```
control_block_ptr = other->control_block_ptr;
control_block_ptr->refs.fetch_add(1, memory_order_relaxed);
```

Decrement

```
if (!control_block_ptr->refs
      .fetch_sub(1, memory_order_acq_rel)) { // key
  delete control_block_ptr;
}
```

Increment can be `relaxed` (not a publish operation, no one depends on this increment). Decrement can be `acq_rel` (both acq+rel necessary, probably sufficient).

Let's analyse the wrong one that `fetch_sub` uses pure `memory_order_release`.

VS2012's bug with ARM architecture, x86-64 is much stronger it's OK with it.

```
if (!control_block_ptr->refs
      .fetch_sub(1, memory_order_release)) { // buggy
  delete control_block_ptr;
}
```

e.g, wrong memory order, pure `release`, consider 2 threads need to decrement

```
// Thread 1, 2->1                        |    // Thread 2, 1->0
// A: use of object                      |    :::
if (!control_block_ptr->refs              |    if
(!control_block_ptr->refs
      .fetch_sub(                        |         .fetch_sub(
        1, memory_order_release)) {      |             1,
memory_order_release)) {
  // branch not taken                    |       delete
control_block_ptr; // B
}                                       |    }
```

The explanation given by Herb Sutter

- *No acquire/release => no coherent communication guarantee that thread 2 sees thread 1's writes in the right order. To thread 2, line A could appear to move below thread 1's decrement even though it's a release(!).*
- *Release doesn't keep line B below decrement in thread 2.*

The second reason is obvious, because it's a `release-operation`, it's free to reorder that delete floating up.

It's hard to understand the first one for the first time, to there are 2 questions:

1. how can decrementing from 2 to 1 goes before line A
2. how can line A come after line B

To figure it out, we need to some transformations that compiler and CPU will do with the pure `release` semantics.

Transformation 1, break thread 1 into pieces (the load and store is for demonstration, they actually may be a single instruction on some hardwares)

```
// Thread 1, 2->1


// A: use of object

load ref_cnt
decrement ref_cnt 2->1
store ref_cnt
"release fence"
if (ref_cnt == 0) { // ref_cnt in register
  // branch not taken
}
```

Transformation 2, reordering happens to decrementing of ref_cnt

```
// Thread 1, 2->1

load ref_cnt
decrement ref_cnt 2->1 // reordered
store ref_cnt

// A: use of object

"release fence"
if (ref_cnt == 0) { // ref_cnt in register
  // branch not taken
}
```

Transformation 3, break thread 2 into pieces

```
// Thread 2, 1->0
:::
load ref_cnt
decrement ref_cnt 1->0
store ref_cnt
"release fence"
if (ref_cnt == 0) { // ref_cnt in register
  delete control_block_ptr; // B
}
```

Transformation 4, reordering happens to that delete

```
// Thread 2, 1->0
:::
load ref_cnt
decrement ref_cnt 1->0
store ref_cnt
if (ref_cnt == 0) { // ref_cnt in register
  delete control_block_ptr; // B
}
"release fence"
```

Transformation 2 and 4 are both legal forms according to the standard:

N3337 29.3.2

*An atomic operation A that performs a release operation on an atomic object M synchronizes with an atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.*

in other words, by cppreference:

*All writes in the current thread are visible in other threads that acquire the same atomic variable, and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic*

Contrast transformation 2 with 4, vertical direction is the execution order.

```
// Thread 1, 2->1                         |        // Thread 2, 1->0
load ref_cnt                             |        :::
decrement ref_cnt 2->1                    |
store ref_cnt                            |
                                       |         load ref_cnt
                                       |         decrement ref_cnt
1->0
                                       |         store ref_cnt
                                       |         if (ref_cnt == 0)
{ // reg
                                       |           delete
control_block_ptr; // B
// A: use of object                       |           }
"release fence"                          |        "release fence"
if (ref_cnt == 0) { // reg                 |
  // branch not taken                     |
}                                      |
```

figure , `release` doesn't keep `synchronizes-with` relation

**Thread 1's use of object comes after thread 2's delete of that object, bang!** This is the situation Herb describes, when 2 threads try to decrement, there actually is a data dependency, one must "wait" for, synchronizes-with another, if

using `release` only, there is no such a `synchronize with` semantics.

With `synchronizes-with`, there won't be any transforms like figure given above

with `fetch_sub(memory_order_acq_rel)`.

With semantics of "acquire + release", "A: use of object" cannot move down into or past the "critical section", and "delete control_block_ptr" cannot move up into or past "critical section", problem solved.

```
// Thread 1, 2->1                         |       // Thread 2, 1->0
// A: use of object //cannot move down|
"acquire & release fence"                  |
load ref_cnt                             |
decrement ref_cnt 2->1                    |        :::
store ref_cnt                synchronizes-with
```

```
"acquire & release fence"  ----------------->  "acquire &
release fence"
                                   |       load ref_cnt
                                   |       decrement ref_cnt
1->0
                                   |       store ref_cnt
                                   |       "acquire & release
fence"
if (ref_cnt == 0) {                  |       if (ref_cnt == 0)
{ // can not move up
  // branch not taken                |         delete
control_block_ptr; // B
}                                  |       }
```

Note: Putting the term “release fence” and “acquire fence” in double quote is to distinguish with the standalone fence in C++11.

### 11.5 rwlock of glibc

From version 2.28 on, rwlock in glibc is implemented with lock-free technique, which may be faster than shared lock provided by C++ standard lib (since c++14), although there are still some low level locks required.

This is a very difficult lock-free algorithm, if you understand all the purposes of the atomic operations, you may be good to say that you understand lock-free programming well.:smirk:

glibc 2.28 release https://www.sourceware.org/ml/libc-alpha/2018-08/msg00003.html

`nptl/pthread_rwlock_common.c` contains most of the implementation.

---

# 12 Recap and conclusion

This post first introduces the hardware architecture cache and cache line, which may introduce OoO, and then discusses about runtime and compile-time reordering introduced by optimization, and later on, we discusses about how to control the reordering. Hence, we introduce the hardware and software memory model. After

Introduction of MM, we talk lots of c++ memory ordering, and MM of C++11. And finally we bring up the lock-free programming which is the real application of MM.

Hardware, compiler and C++ are complicated and powerful, we may not know all the stuffs they've done for us, but we need to know the principle, that's what this post tries todo.

Modern C++, especially C++11, abstracts the memory model very well, the concepts and potability of `<atomic>` are very good, helps C++ users a lot, it's very hard to do that, but C++ standard committee actually did, many thanks to them!

We talked lock-free programing at last, and found it's a very tough task to do general tasks with lock-free technique, we have to do a lot trade-offs and be very very very careful when we are doing lock-free programming.

Finally,

*With Great Power Comes Lots Of Fun!*

---

## 13 Useful resources

### 13.1 Static materials

*an-introduction-to-lock-free-programming*
*memory-ordering-at-compile-time*
*memory-barriers-are-like-source-control-operations*
*acquire-and-release-semantics*
*acquire-and-release-fences*
*weak-vs-strong-memory-models*
*the-synchronizes-with-relation*
*double-checked-locking-is-fixed-in-cpp11*
*acquire-and-release-fences-dont-work-the-way-youd-expect*
*the-purpose-of-memory_order_consume-in-cpp11*
*can-reordering-of-release-acquire-operations-introduce-deadlock*

Jeff Preshing's posts of "lock-free programing" series, if you are new to lock-free programming or software/hardware memory ordering, read them in the given order, I can ensure that it may enlighten you greatly.

And you are welcomed to discuss with me about any

*Intel Software Developer Manuals Combined Volumes: 3a, 3b, 3c and 3d - System programming guide*

Volume 3 §8.1 and §8.2 are the same important reference, they show the Intel x86-64 family processors' reordering detailed specification, which is worth reading.

*GCC's reordering of read/write instructions*

what is the benefit of instruction at compile time and runtime

*CPU cache*
*CPU cache coherence*
*intel coherence protocol*
*hyper threading*
*intel-introduction-to-x64-assembly*

The above materials are about hardware level that related to memory order.

*double-checked locking is broken for java*

This post shows and analyses the broken DLCP for java in compile-time, the constructor is inlined by javac.

However, I do not see any thing wrong with gcc, the generated assembly code is just fine, the memory order is correct… Maybe explicit or compiler optimization `inline` is needed, that a complicated case.

*c++ working drat*

Here is the C++ Standard Draft Sources, check it for the latest C++ working draft.

*C++ working draft N3337*

This is the latest publication of C++11 working draft.

*Single Threaded Memory Model - an example of out of thin air* *Optimization of conditional access to globals: thread-unsafe?*

This mailing list about the conditional access to memory, which covers the discussion about out-of-thin-air, and it may affect sequentially consistency when compiler optimize this kind of conditional access.

*The C++ Programming Language 4th Edition by Bjarne Stroustrup*

Chapter 41 talks about concurrency, memory model and atomic, which is really helpful for understanding these concepts.

The purpose/definition of memory model is defined in §41.1.

*obstruction-freedom*

*Lock-Free and Wait-Free, definition and examples*

An article puts definitions of blocking, lock-free, wait-free and obstruciton-free all together.

Blocking 1. Blocking 2. Starvation-Free Obstruction-Free 3. Obstruction-Free Lock-Free 4. Lock-Free (LF) Wait-Free 5. Wait-Free (WF) 6. Wait-Free Bounded (WFB) 7. Wait-Free Population Oblivious (WFPO)

*rust memory ordering*

Introduces the memory ordering in rust.

*lock-free statck written with C11*

This is a versioning solution to solve the ABA problem of lock-free singly-liked list.

*lock-free singly-linked list written with C++11*

This is an attempt to implement a lock-free singly-linked list using hazard pointer and reference counting technique.

*lock-free data structure lib in C*

Learn from others! This lock-free lib is written in C, may lack of portibility.

*linus' comment on store buffer* *store buffer*

Some disscusion on store buffer.

*A primer on memory consistency and cache coherence*

一本书，有 pdf 版本. 从硬件设计的角度阐述如何设计原子指令，硬件的内存模型是什么，如何保证内存模型的 正确性

*C++ Concurrency in Action, 2nd Edition*

一本书，有 pdf 版本. C++ 并行编程，有锁无锁，内存模型讲的比较清楚. 附带了一些例子.

*MESI cache coherence protocol*

*"Memory System (Memory Coherency and Protocol)" (PDF). AMD64 Technology. September 2006.*

## 13.2 Videos and talks

*CppCon 2014: Herb Sutter "Lock-Free Programming" 1/2*
*CppCon 2014: Herb Sutter "Lock-Free Programming" 2/2*

Herb Sutter's talk about lock-free programming, part 1 introduce some fundamentals, and part 2 is mainly about some concrete examples like: lock-free single list with push and pop and some analysis. Herb Sutter also implicitly talks

*C++ and Beyond 2012: Herb Sutter - atomic Weapons 1 of 2*
*C++ and Beyond 2012: Herb Sutter - atomic Weapons 2 of 2*

Herb Sutter's talk about atomic, part 1 is mainly about the fundamentals that what is memory order and why they exists, and what is `acquire and release semantics`, what does the compiler do for optimization related with instruction compile-time reordering.

Part 2 is mainly about the optimization that the compiler does about memory order. In this part Herb shows the emitted assembly code for different architecture CPUs under different circumstances, which can help understanding the software `acquire and release semantics` on hardware level.

*CppCon 2014: Jeff Preshing "How Ubisoft Develops Games for Multicore - Before and After C++11"*

Jeff's talk about atomic lib of C++11.

*ACCU 2017 - Atomic's memory orders, what for? - Frank Birbacher*

This talk is mainly about "synchronize-with" relation explanation.

*The C++ Memory Model - Valentin Ziegler @ Meeting C++ 2014*

This talk is about memory model of C++11, the first half explains "Sequential Consistency" in CPP memory model very well, the second half is not that informative.

*CppCon 2015: Michael Wong "C++11/14/17 atomics and memory mode*

This talk introduce the memory model of C++11/14/17 on a high level. The speaker introduces a lot of concepts and definitions of design of memory model of C++11/14/17, like:

1. sequential consistency
2. what is memory model.
3. synchronizes-with, sequenced-before, happens-before
4. etc. it's worth watching though the content arrangement seems a little messy.

*CppCon 2017: Fedor Pikus "C++ atomics, from basic to advanced. What do they really do?"*

This talk gives very clear explanation of CAS weak/strong, Fedor uses DCLP to explain CAS implementation, which is impressive. And he also talks about performance compared to non-atomic operations and mutex operations.