原子内存访问在 C/C++11 中的语义和实现

简介

C11 和 C++11 (以下简称C/C++11) 规定了原子操作对内存的访问的一致性机制，从弱到强有： relaxed, acquire/release, sequentially consistent。然而由于不同硬件，更精确地说，计算机体系结构的不同，可能会有不同的内存访问模型，而这导致了 C/C++11 的原子操作在各个平台上的实现也不同，甚至出现问题，本报告围绕 Ori Lahav et al. Repairing Sequential Consistency in C/C++11 提出的问题, 从上而下地解释 C/C++11 到各架构处理器的内存访问机制，说明 C/C++11 在 IBM Power 架构上的实现为何是有缺陷的，以及已经提出的解决办法。

C/C++11 标准中的内存访问

由于 C11 和 C++11 在这一部分的标准基本类似，现仅说明 C++11 部分的内存访问。

操作的语法

C++11 中新增了 `<atomic>` 库，其中包括了原子类型，以及对其操作。

```
std::atomic<int> a;
```

对每一个原子操作，都需要指定一个访问顺序，访问顺序的定义如下：

```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
} memory_order;
```

在进行原子操作的时候，需要指定其内存顺序 (默认为 `memory_order_seq_cst`)

```
int x = a.load(memory_order_acquire); // load a's value
a.store(42, memory_order_relaxed); // store 42 to a
int y = a.fetch_add(3, memory_order_seq_cst); // fetch a's value, then add 3 to a
assert(y == 42); // do not fire
```

各种 memory_order 及其含义

1. relaxed

这是最弱的一种约束，仅保证操作自身的原子性，而对原子操作之间的顺序没有要求。编译器可以调整 relaxed 操作之间的顺序，也允许硬件乱序执行 relaxed 操作。

2. release / acquire

这种约束较 relaxed 强，其一定程度上约定了访存的顺序。

如果一个 load-acquire 的结果是一个 store-release 的结果，或者是这个 store-release 开头的 release 序列中写的结果，则该 release 之前的操作对执行 load-acquire 的线程来说，在 load 操作结束后均保证为可见的。

因此，任意读写不能交换到一个 load-acquire 操作之前，或者一个 store-release 操作之后执行。

一种实现是一个线程在 release 时将本地的写都广播到所有线程。

3. sequentially consistent

Sequentially consistenty 是 C/C++11 atomic 提供的最强的访问一致性。

如果一个 store 是 seq_cst，则其会进行 release。如果一个 load 是 seq_cst， 则其会进行 acquire。

在此之外，所有的 seq_cst 操作对发生前后关系构成一个全序集，所有线程下该关系都一致。

4. consume

consume 和 acquire 相近，下面的内存模型不提及 consume ，因此在这里不具体描述而导致问题复杂化。

硬件内存模型

强内存模型和弱内存模型

虽然 C/C++11 同一规定了内存的访问次序，但在不同的硬件上，其实现有很大的差异。其差异主要体现在内存模型的不同上。

内存模型描述了不同线程与内存的交互以及其数据的共享，由计算机的架构决定。在已知内存模型的情况下，编译器可以对程序进行重要的优化，而在未知的情况下，进行这种优化可能会导致数据竞争的情况出现。

而根据内存对读写顺序的限制，又可以大致将内存模型分为两种。

1. 强内存模型：对读写的顺序有很大的限制，硬件几乎不能对访存重排序
2. 弱内存模型：对读写的顺序限制较小

强内存模型

处理器架构如 x86，AMD64，运行在 TSO 模式（TSO 指 total store order）的 SPARC为强内存模型。其主要特点为：

- 除了读可以调整到不同地址的写之前以外，不允许原子读和原子写顺序的交换
- 对所有线程来说，所有写有一个一致的全序。

强内存模型例子：x86-TSO

x86-TSO 是 Peer Sewell et al. 提出的面向程序员的 x86 多线程处理器的一个内存模型。其主要特点为：

1. 所有线程都有一个 FIFO 写缓存器。任何线程的读操作，如果其地址在写缓冲区中存在，则一定读取其中最近的一次写，如不存在，则向共享内存发起读。
2. mfence 操作可以清空写缓存。
3. 标记 LOCK 的指令要求程序获得全局的锁，并且在该指令结束时，清空写缓冲区并释放锁。

4. 一个线程可以在除别的线程获得锁以外的任何时刻，将缓冲的写操作写至共享内存。

C/C++11 在 x86-TSO 下的实现

在如 x86-TSO 这样的强内存模型下，实现 C/C++ 原子库的功能较为容易。

在 C/C++11 mappings to processors 中记录了通常翻译到 x86 的代码。

C/C++11 Operation	x86 implementation
Load Relaxed:	MOV (from memory)
Load Consume:	MOV (from memory)
Load Acquire:	MOV (from memory)
Load Seq_Cst:	MOV (from memory)
Store Relaxed:	MOV (into memory)
Store Release:	MOV (into memory)
Store Seq Cst:	(LOCK) XCHG OR MOV (into memory),MFENCE

弱内存模型

处理器架构如 IBM Power，ARM，Itanium使用的是弱内存模型。其主要特点为：

- 除了有依赖的原子读和原子写不能交换，以及 memory fences 以外，原子读和写均可以交换次序

C/C++11 在 Power 下的实现

C/C++11 Operation	Power implementation
Load Relaxed:	ld
Load Acquire:	ld;
Load Seq Cst:	ld; hwsync
Store Relaxed:	st
Store Release:	lwsync; st
Store Seq Cst:	lwsync; st; hwsync

C/C++11 在弱内存模型下实现的问题

Ori Lahav et al. (2017) 指出，C/C++11 的内存模型在 IBM Power, ARMv7 这些弱内存模型上目前的实现是有问题的。

首先是 C/C++ 要求的 SC (sequentially consistency) 过强，以至于在这些机器上的现有实现在混合 acquire 和 seq_cst 访问时，其性质会被破坏。如下面这个例子(Manerkar et al., 2016)：

Thread 0	Thread 1	Thread 2	Thread 3
$a := x_{acq}$ // 1 $c := y_{sc}$ // 0	$x :=_{sc} 1$	$y :=_{sc} 1$	$b := y_{acq}$ // 1 $d := x_{sc}$ // 0

这里用下标表示内存访问类型，注释为读操作的结果。$x, y, \ldots$ 为内存，$a, b, \ldots$ 为本地变量。所有变量初始值均为0。

C/C++ 禁止这种结果的发生。而在 Power 架构下这种情况是可能的。

C/C++ 禁止的判断

C/C++11 中有如下规则：

- *happens-before* 顺序：在不考虑 consume 的情况下（事实上当前草稿不建议使用 consume ）， *happens-before* 的定义等同于 *strongly happens-before*，说求值 A *strongly happens before* 求值B，应满足以下任意一个条件：
  1. A *sequenced before* B，即在一个线程内， A 的求值顺序先于 B。
  2. B *synchronizes-with* A，即 B 是 load-acquire，且读取的结果为 A 写的结果，或 A 为首的 release sequence 中写的结果。
  3. A *strongly happens before* X，且 X *strongly happens before* B
- 所有标记为 SC 的事件应存在一个唯一的全序（唯一指各线程上都有该全序），该全序与 happens-before 以及每一个内存地址上的 SC 写的顺序无矛盾。
- 标记为 SC 的读会读取：

1. 如果该读之前（在上述 SC 全序的关系下）有对同一内存地址的 SC 写，那么会读下列两者之一：
   - SC 的全序之中，在该读之前距其最近的 SC 写
   - 一个非 SC 的写，这个写不在对该地址的 SC 写之前发生(happen before)
2. 如果没有，则为某非 SC 的写操作的结果

使用上述的规则，可以发现：

1. happens before 关系：$x :=_{sc} 1$ -> $a := x_{acq}$ -> $c := y_{sc}$, $y :=_{sc} 1$ -> $b := y_{acq}$ -> $d := x_{sc}$
2. 各个内存位置的读写顺序： $c := y_{sc}$ -> $y :=_{sc} 1$ -> $b := y_{acq}$, $d := x_{sc}$ -> $x :=_{sc} 1$ -> $a := x_{acq}$
3. 由 1.2. 得到的 SC 全序产生了一个环，因此是不允许的

Power 架构下的实现

这里考虑 trailing sync ，即 SC 操作翻译到弱内存模型的指令时， sync(Power内即hwsync) 指令在最后。

可以发现 Thread 0 中 a 和 c 之间只有一个 lwsync，该 sync 过弱而使得可能 Thread 0 上看到 x=1, y=0 而同时 Thread 1 看到 x=0, y=1 ，导致矛盾。

如果使用 leading sync ，则 Thread 0 在读取 x=1 之后，会使用 hwsync 使得 x=1 对所有线程可见，因此这个程序不会出问题。但能构造程序， 使得 leading sync 发生类似的问题。

问题原因

具体原因为 release/acquire 使用的 lwsync 过弱，导致不能保证前后操作的顺序性。

解决方法

如果将 release/acquire 的 lwsync 全部换成 hwsync ，则可以解决该问题，然而 hwsync 需要同步所有线程，导致代价十分高昂，release/acquire 也也失去其意义。

Ori Lahav et al.(2017) 提出了一种解决该问题的方法 (S1fix)，即削弱对 SC 顺序的一致性要求，从原来的与 *sequenced-before* 和 *happens-before* 的闭包一致，减弱至和 $(sb \cup sb; hb; sb \cup hb|_{loc})$ (sb 即 sequence before，hb 为一个内存地址上读写的 happen before) 一致，即 SC 中两个事件不需要和仅由 *happens-before* 构成的闭包的顺序一致。

第二种解决方案下，修改后的 C/C++ 标准（论文中称为 RC/C++11）允许 ARM/IBM Power 的这种行为。

参考文献

x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors std::memory_order - cppreference.com C++ standard draft n4750 C/C++11 mappings to processors Repairing Sequential Consistency in C/C++11 A Tutorial Introduction to the ARM and POWER Relaxed Memory Models