内存屏障和内存模型

2020.06.10 SF-Zhou

1. 内存屏障

从维基百科中摘录的定义：

*A **memory barrier**, also known as a **membar**, **memory fence** or **fence instruction**, is a type of barrier instruction that causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction.*

定义来看，内存屏障是用来强制约束屏障指令前后内存操作顺序。现代 CPU 会采用乱序执行、流水线、分支预测、多级缓存等方法提高性能，但这些方法同时也影响了指令**执行**和**生效**的次序。这种乱序对单线程执行是无影响的，执行的结果与原指令顺序执行保持一致，但在多线程环境下则会造成非预期的行为，举个例子（参考文献 2 Example 8-3）：

```
; initially x = y = 0

; processor 0
mov [_x], 1
mov r1, [_y]

; processor 1
```

```
mov [_y], 1
mov r2, [_x]

; r1 = r2 = 0 is allowed
```

如果严格按顺序执行， r1 和 r2 至少有一个会是 1；但如果有乱序发生， r1 和 r2 的赋值中 x 和 y 的读取发生在 x 和 y 的赋值生效前，那就可能产生 r1 = r2 = 0 的情况。这时候就需要使用内存屏障来约束指令的执行和生效的次序，解决硬件层面的指令重排序和可见性问题。

以 Intel 的 x86 系列 CPU 为例，可以使用 mfence 指令实现内存屏障。 mfence 前后的指令会严格以 mfence 为界线， mfence 之前的读写指令会在界线之前完成并全局生效，之后的指令不会重排到界线之前执行。使用 mfence 指令修改上面的例子：

```
; initially x = y = 0

; processor 0
mov [_x], 1
mfence
mov r1, [_y]

; processor 1
mov [_y], 1
mfence
mov r2, [_x]

; r1 = r2 = 0 is not allowed
```

除了 mfence 外， lock add 、 xchg 等指令也可以实现内存屏障的功能，不过一般情况下并不推荐在代码中直接使用这些汇编指令。

2. 内存模型

高级编程环境一般有自己的高级内存模型并定义了内存可见性语义，其内存模型是建立在硬件内存模型之上的一层抽象，见参考文献 3。高级编程环境一般会提供内存序、互斥量、信号量等同步原语，通过编译器翻译到硬件层的汇编指令，这种情况下一般不需要显式地使用内存屏障指令。

以 C++ 为例，C++ 11 之后提供了原子量和六种内存序来解决多线程下的内存一致性问题。建议先耐心地看完参考文献 5 中 std::memory_order 的描述。下面是笔者的一些理解：

1. std::memory_order_relaxed 对原子量只保证原子性；
2. Release-Acquire 语义，当 x.load(std::memory_order_acquire) 读到 x.store(r, std::memory_order_release) 的修改时，前者后面的指令必定可以读到后者前面指令的修改；
3. Sequentially-Consistent 语义，包含 Release-Acquire 语义，并且所有线程都以相同的顺序观察到所有 std::memory_order_seq_cst 的修改；
4. std::memory_order 是通过限制本线程内的代码执行和生效顺序，来解决多线程下的内存一致性/可见性问题。

高级内存可见性语义与硬件内存屏障语义并不是一一对应的。以 x86 系列 CPU 为例，由于其本身已经具备了强内存一致性语义，C++ 提供的大部分内存序在编译后都会被省略

掉，仅会阻止编译器的重排。x86 只会在 Store-Load 情况下产生可见性乱序，举个例子（修改自参考文献 7）：

```
#include <bits/stdc++.h>

void threadfun(std::atomic<int> &cnt0, std::atomic<int> &cnt1,
               std::atomic<int> &x, std::atomic<int> &y, std::atomic<int> &r) {
  while (true) {
    // barrier, wait cnt0 increase
    while (cnt0 == cnt1) {
    }

    x.store(1, std::memory_order_release);  // need std::memory_order_seq_cst
    r.store(y.load(std::memory_order_acquire), std::memory_order_release);

    cnt1.fetch_add(1, std::memory_order_acq_rel);
  }
}

int main() {
  alignas(64) std::atomic<int> cnt0{0};
  alignas(64) std::atomic<int> cnt1{0};
  alignas(64) std::atomic<int> cnt2{0};
  alignas(64) std::atomic<int> x{0};
  alignas(64) std::atomic<int> y{0};
  alignas(64) std::atomic<int> r1{0};
  alignas(64) std::atomic<int> r2{0};
```

```
  std::thread thr1(threadfun, ref(cnt0), ref(cnt1), ref(x), ref(y), ref(r1));
  std::thread thr2(threadfun, ref(cnt0), ref(cnt2), ref(y), ref(x), ref(r2));

  while (true) {
    x.store(0, std::memory_order_release);
    y.store(0, std::memory_order_release);

    cnt0.fetch_add(1, std::memory_order_acq_rel);
    // barrier, wait cnt1 and cnt2 increase
    while (cnt1 != cnt0 || cnt2 != cnt0) {
    }

    if (r1 == 0 && r2 == 0) {
      std::cout << "store-load reorder" << std::endl;
      break;
    }
  }

  return 0;
}
```

代码和第一节中的汇编例子是一致的。即使全部使用了原子量和 Release-Acquire 语义，这里依然会存在 Store-Load 乱序。在 C++ 语言层面上这段代码也是错的，可以仔细揣摩下。下面是笔者的一些观察：

1. 当 cnt1==cnt0 时，根据可见性的递推， r1 必定可以读到线程中的修改，其他多线程通信中的可见性问题也与之类似；
2. 在 x86 环境下上述代码 x.store 和 y.load 之间不会产生 mfence 或其他内存屏障指

令，硬件层面也说明上面的代码是错的；

3. std::atomic::fetch_add 会编译成 lock add，由于自带了内存屏障语义，在 x86 环境下使用任何内存序编译结果都是一样的；

4. std::atomic::load 由于 x86 的强一致性，使用任何内存序编译结果都不会产生内存屏障指令，但使用非 relaxed 内存序时会阻止编译器的重排。

上面这段代码有几种修改方法：

1. 对 x.store 使用 std::memory_order_seq_cst；
2. 在 x.store 和 y.load 之间增加 std::atomic_thread_fence(std::memory_order_seq_cst)。

3. 应用举例

使用原子量实现一个基于 Ring-Buffer 的 MPSC 无锁队列：

```
#include <atomic>
#include <cstdint>
#include <memory>
#include <thread>
#include <vector>

template <typename T>
class LockFreeQueue {
 public:
  enum {
```

```
    kEmpty = 1,
    kFull = 2,
    kConflict = 3,
    kInvalid = 4,
  };

public:
  explicit LockFreeQueue(uint64_t capacity)
      : capacity_(capacity),
        items_(std::make_unique<std::unique_ptr<T>[]>(capacity)) {}

  uint64_t Size() const {
    auto t = tail_.load();
    auto h = head_.load();
    return h - t;
  }

  bool Empty() const { return Size() == 0; }

  bool Full() const { return Size() >= capacity_; }

  int PopByOneThread(std::unique_ptr<T> *item) {
    auto t = tail_.load();
    auto h = head_.load();

    if (h == t) {
      return kEmpty;
    }

    auto &pop = items_[t % capacity_];
```

```
    if (pop == nullptr) {
      return kConflict;
    }
    *item = std::move(pop);


    tail_.fetch_add(1);
    return 0;
  }

  int PushByMultiThread(std::unique_ptr<T> *item, int retry_times = 5) {
    if (item->get() == nullptr) {
      return kInvalid;
    }
    for (int i = 0; i < retry_times || retry_times == -1; ++i) {
      auto t = tail_.load();
      auto h = head_.load();

      if (h - t >= capacity_) {
        return kFull;
      }

      if (head_.compare_exchange_strong(h, h + 1)) {
        item->swap(items_[h % capacity_]);
        return 0;
      }
    }
    return kConflict;
  }

private:
```

```
  std::atomic<uint64_t> head_{0};
  std::atomic<uint64_t> tail_{0};

  const uint64_t capacity_;
  std::unique_ptr<std::unique_ptr<T>[]> items_;

};

int main() {
  constexpr int N = 1000000;
  constexpr int M = 4;
  LockFreeQueue<int> queue(N * M);

  std::vector<std::thread> threads;
  for (int i = 0; i < M; ++i) {
    threads.emplace_back([&] {
      for (int j = 0; j < N; ++j) {
        auto item = std::make_unique<int>(j);
        while (queue.PushByMultiThread(&item) != 0) {
        }
      }
    });
  }

  std::unique_ptr<int> item;
  for (int i = 0; i < N * M; ++i) {
    while (queue.PopByOneThread(&item) != 0) {
    }
  }

  for (auto &thread : threads) {
```

```
    thread.join();
  }
  return queue.Size();
}
```

该队列有一个隐含的假设，即对指针长度的数据读写是原子的，在 x86 上该假设是满足的。通过 CAS 可以实现多线程的 Push，Pop 部分也可以改为 CAS 实现 MPMC 队列。

接下来分析 C++ 层面是否可以优化 std::memory_order 。 fetch_add 和 compare_exchange_strong 操作均需要被接下来的读取立即读到，所以这里需要用默认的 std::memory_order_seq_cst； tail_.load 和 head_.load 立即读到上次的修改会更好，所以也使用 std::memory_order_seq_cst；这样保持使用默认参数即可。

在 x86 平台上，原子量的 load 操作使用非 std::memory_order_relaxed 的编译结果是一致的； fetch_add 和 CAS 自带内存屏障也没有优化的必要和可能，这样看使用默认参数就已经足够好了。

4. 非对称内存屏障

某些并发算法中会存在简单路径和复杂路径，并且需要内存屏障完成同步。如果全部使用 std::memory_order_seq_cst ，会导致简单路径时间成本过高。这时候就可以用上非对称内存屏障，在简单路径中快速执行，将同步的复杂度转移到复杂路径中。以 x86 平台为例，简单路径中可只保证编译器指令不重排，复杂路径中可使用 mprotect 强制所有线程内存一致，可参考 folly 的实现。这里举个例子（在线执行）：

```
#include <atomic>
```

```
#include <bits/stdc++.h>
#include <sys/mman.h>

#include <cassert>

static void *CreatePage() {
  auto ptr = mmap(nullptr, 1, PROT_READ, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  assert(reinterpret_cast<ssize_t>(ptr) != -1);

  // Optimistically try to lock the page so it stays resident. Could make the
  // heavy barrier faster.
  auto r = mlock(ptr, 1);
  if (r != 0) {
    // Do nothing.
  }

  return ptr;
}

void HeavyBarrier() {
  static auto page = CreatePage();

  // This function is required to be safe to call on shutdown, so we must leak
  // the mutex.
  static std::mutex mutex;
  std::lock_guard<std::mutex> lg(mutex);

  // We want to downgrade the page while it is resident. To do that, it must
  // first be upgraded and forced to be resident.
  int r = mprotect(page, 1, PROT_READ | PROT_WRITE);
  assert(r != -1);
```

```
  // Force the page to be resident. If it is already resident, almost no-op.
  *static_cast<char *>(page) = 0;

  // Downgrade the page. Forces a memory barrier in every core running any of

  // the process's threads. On a sane platform.
  r = mprotect(page, 1, PROT_READ);
  assert(r != -1);
}

inline void LightBarrier() { asm volatile("" : : : "memory"); }

void threadfun(std::atomic<int> &cnt0, std::atomic<int> &cnt1,
               std::atomic<int> &x, std::atomic<int> &y, std::atomic<int> &r,
               int barrier_type) {
  while (true) {
    // barrier, wait cnt0 increase
    while (cnt0 == cnt1) {
    }

    x.store(1, std::memory_order_release);
    if (barrier_type == 0) {
      LightBarrier();
    } else {
      HeavyBarrier();
    }
    r.store(y.load(std::memory_order_acquire), std::memory_order_release);

    cnt1.fetch_add(1, std::memory_order_acq_rel);
  }
```

```
}

int main() {
  alignas(64) std::atomic<int> cnt0{0};
  alignas(64) std::atomic<int> cnt1{0};

  alignas(64) std::atomic<int> cnt2{0};
  alignas(64) std::atomic<int> x{0};
  alignas(64) std::atomic<int> y{0};
  alignas(64) std::atomic<int> r1{0};
  alignas(64) std::atomic<int> r2{0};

  std::thread thr1(threadfun, ref(cnt0), ref(cnt1), ref(x), ref(y), ref(r1), 0);
  std::thread thr2(threadfun, ref(cnt0), ref(cnt2), ref(y), ref(x), ref(r2), 1);

  while (true) {
    x.store(0, std::memory_order_release);
    y.store(0, std::memory_order_release);

    cnt0.fetch_add(1, std::memory_order_acq_rel);
    // barrier, wait cnt1 and cnt2 increase
    while (cnt1 != cnt0 || cnt2 != cnt0) {
    }

    if (r1 == 0 && r2 == 0) {
      std::cout << "store-load reorder" << std::endl;
      break;
    }
  }

  return 0;
```

```
}
```

显然这里 HeavyBarrier 的成本是比较高的，所以使用上需要慎重，类似 Hazard Pointer 这种简单路径非常频繁就比较适合。好消息是 Linux 4.x 之后提供了 membarrier 系统调用效率更高，可以替代掉这里的 HeavyBarrier 。该函数的文档中也有关于非对称内存屏障的说明，见参考文献 10。

References

1. "Memory Barrier", *Wikipedia*
2. "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1", *Intel*
3. "std::memory_model", *C++ Reference*
4. "std::atomic", *C++ Reference*
5. "std::memory_order", *C++ Reference*
6. "volatile与内存屏障总结", *郑传军*
7. "X86/GCC memory fence的一些见解", *饶萌*
8. "如何理解 C++11 的六种 memory order？", *陈文礼*
9. "P1202: Asymmetric fences", *David Goldblatt*
10. "membarrier(2)", *Linux manual page*

SF-Zhou Jun 16, 2020 Owner

如果你写的代码只在 x86 上执行，那么除了 `std::atomic<T>::store` 函数外，其他的 `std::memory_order` 都可以使用默认的，性能不会有区别。

↑ 1 ☺ 0 replies

Write Preview Aa

Sign in to comment

Styling with Markdown is supported Sign in with GitHub