Document number: P1202R0
Date: 2018-10-06 (pre-San Diego)
Reply-to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: SG1

P1202: Asymmetric fences

Overview

Some types of concurrent algorithms can be split into a common path and an uncommon path, both of which require fences (or other operations with non-relaxed memory orders) for correctness. On many platforms, it's possible to speed up the common path by adding an even stronger fence type (stronger than `memory_order_seq_cst`) down the uncommon path. These facilities are being used in an increasing number of concurrency libraries. We propose standardizing these asymmetric fences, and incorporating them into the memory model.

Background

Many synchronization algorithms have at their core a store-load fence: Hazard Pointers, most userspace RCU implementations, some types of work-stealing queue, RMW-less mutexes, etc.

In some cases, the fence is required down a common path, only to guard against a race with an uncommon path:

- In hazard pointers, the hazptr_holder `try_protect()` path is common, and the reclamation path it protects against is uncommon.
- In RCU, the `rcu_reader` constructor path is common, and the `rcu_synchronize()` path is uncommon.
- In work-stealing queues, the owner-dequeue path is common, and the steal-dequeue path is uncommon.
- Load-store mutexes are not commonly used. However, with variants of the asymmetric fences described later on, they are the usual implementation technique for biased locking, in which the biased-thread locking path is common, and all non-biased thread locking paths are uncommon.

This puts library authors in a frustrating position: they have to insert store-load fences (typically the most costly type) down common paths, to ensure safety against a rare race down already-slow paths. Existing alternatives within the memory model are usually abstraction-hostile (such as requiring library users to periodically call a `quiesce()` function, register and unregister themselves when active, etc.).

An alternative to fences on the fast path

Some platforms provide ways to provide the effects of a memory fence in other threads through out-of-band mechanisms, without actually inserting one. This lets code like the following store-buffering example work correctly:

```
// Globals:
atomic_int x{0}, y{0};
int r1, r2;

// Fast path:
x.store(1, memory_order_relaxed);
asymmetric_thread_fence_light(); // Like atomic_signal_fence;
                                 // only inhibits compiler reordering
r1 = y.load(memory_order_relaxed);

// Slow path:
y.store(1, memory_order_relaxed);
asymmetric_thread_fence_heavy(); // Stronger than
                                 // atomic_thread_fence(memory_order_seq_cst);
r2 =  x.load(memory_order_relaxed);

// Happens after both fast path and slow path are complete.
assert(!(r1 == 0 && r2 == 0)); // Never fails.
```

The specific implementation of `asymmetric_thread_fence_heavy()` is platform-dependent, and requires at least some level of OS support if it is to improve fast-path performance. The Linux `membarrier()` system call, or Windows `FlushProcessWriteBuffers()` can provide this support (which is implemented either by waiting for all cores to end a scheduling quantum, or sending inter-processor interrupts to other cores).

Some libraries have used less well-supported mechanisms for achieving the same effect (note that some of these are not guaranteed to work, or even disavowed by the appropriate maintainers):

- Relying on `munmap()` using an IPI to trigger TLB shootdowns
- Pinning the calling thread to each core in succession
- Sending a signal to all threads in the process, and waiting for a reply from your signal handler
- Performing an unaligned atomic RMW that spans multiple cachelines, forcing an interconnect quiescence period

The reason to go to such lengths is fast-path performance. In Facebook's open-source folly library, when using the `membarrier()` system call on the writer side (and only a compiler barrier on the reader side), we obtain costs of about 2ns for `hazptr_holder::protect()` and `reset()`, or `rcu_reader` construction and destruction (measured on a 2.5Ghz Haswell). When replacing these asymmetric barriers with an `atomic_thread_fence(memory_order_seq_cst)`, the cost jumps to 19ns (for hazard pointers) and 32ns (for RCU). These numbers are all from microbenchmarks; in real workloads, the cost difference can be even higher.

This technique (or some closely related one) is used in Hotspot, the .NET CLR, liburcu, qemu, the Objective-C runtime, and Grand Central Dispatch, among others.

Let's standardize it

Most of the benefits to standardizing the memory model also apply to asymmetric fencing:

- Portability
- Defined interactions with other portions of the memory model
- Optimizer insight into ordering requirements

There is also the potential for avoiding duplication of work; there are situations in which one thread can "piggyback" off of another's heavy fence. These ought to be rare (since heavy fences themselves should be rare), but can help avoid some pathological behaviors when the slow path is more common than expected.

RCU and Hazard Pointers effectively require asymmetric fences for high-performance implementations. We should try to avoid the situation in which key implementation techniques are not available outside of the standard library.

Formalization

To evaluate different approaches to formalizing these ideas, we choose two criteria:

1. Asymmetric fences should be no stronger than the semantics provided by the underlying implementation.
2. Asymmetric fences should be implementable on platforms that lack underlying OS support, with a cost no more (or at least, not much more) than a plain fence.

The need for item 1 is obvious. Item 2 is necessary to avoid library authors having to implement their own logic for deciding whether or not to use asymmetric or non-asymmetric fences. (Note that on e.g. Linux, the `membarrier()` system call is not universally deployed; the logic for deciding whether or not it can be used is nonportable). If a common path is much faster when switching to a light asymmetric fence some implementations, but much slower on others, users can't portably reason about efficiency, and end up having to write two versions.

To motivate the our suggested approach, we'll look at two simpler ones, and identify where they fail a criterion.

Approach 1

We introduce two new functions:

```
std::asymmetric_thread_fence_heavy()
std::asymmetric_thread_fence_light()
```

We add two rules:

- Heavy fences are also `memory_order_seq_cst` fences.
- For every light fence L and heavy fence H, either H synchronizes with L or L synchronizes with H.

To see how this helps us, consider the store buffering example from above. We want to prove that `r1 == 0 && r2 == 0` cannot occur. Suppose that `r1 == 0`; we must show that `r2 == 1`. We know that everything sequenced before F1 strongly happens before everything sequenced after F2, or vice versa. The latter cannot be the case; it would imply that S2 strongly happens before L1, so that L1's load returns 1. So then S1 strongly happens before L2, so that we have `r2 == 1`.

The problem with approach 1

Perhaps surprisingly, these semantics fail criterion 1; they are not correct with respect to all (most?) of the intended implementation strategies. Broadly, the reason is that the OS-provided heavy fence implementation typically will not stall other threads while the fence is in progress; the issuing thread could synchronize with one light-fencing thread earlier (in physical time) than another.

For a more involved example, consider the following program (with all variables initially zero, using all relaxed atomic operations for shared variables):

`// T0`
`X = 1`
`light_fence()`
`R0 = Y`
	`// T1`
	`Y = 1`
	`heavy_fence()`
	`R1 = Z`
		`// T2`
		`Z = 1`
		`light_fence()`
		`R2 = X`

After these have executed, can we have R0 == 0, R1 == 0, R2 == 0? Yes; with the following sequence of events (in order of physical time, assuming the presence of store-buffers, and a `heavy_fence()` implementation that sends a signal to every thread in the process).

1. T0: X = 1 executes, and the store enters T0's store buffer
2. T0: R0 = Y executes, and sets R0 to 0.
3. T1: Y = 1 executes. It doesn't matter if the store leaves T1's store buffer, since the only read of Y has completed.
4. T1: heavy_fence() sends a signal to T2, and the signal handler completes.
5. T2: Z = 1 executes, and the store to Z enter's T2's store buffer.
6. T2: R2 = X executes, and sets R2 to 0 (the store to X has not yet left T0's store buffer).
7. T1: heavy_fence() sends a signal to T0, and the signal handler completes (flushing T0's store buffer, but too late for it to affect T2). The heavy fence is now done.
8. T1: R1 = Z executes, and sets R1 to 0 (T2's store to Z has not yet left its store buffer).

However, if we used synchronizes-with directly, this outcome would not be allowed; R0 == 0 would imply that T0's light fence synchronizes with T1's heavy fence. Similarly, R1 == 0 implies that T1's heavy fence synchronizes with T2's light fence. Combining these, we would have that T0's store to X should happen before T2's load, and that therefore R2 == 1. (The fact that this example uses store-load sequences rather than a more common pattern is incidental; it's possible to construct a similar one using a three-thread message passing variant. Such an example is trickier to step through, though, since it requires more reordering to get the effect).

Approach 2

We can fix the implementability issue in approach 1 by carving out some of the synchronization, and not providing transitivity across fences. We can give the same semantics as approach 1 viewed pairwise between threads, but without any transitivity, by replacing the second bullet point above with:

- For every light fence L and heavy fence H, one of the following holds:
    a. Every evaluation that is sequenced before L strongly happens before every evaluation that H is sequenced before.
    b. Every evaluation that is sequenced before H strongly happens before every evaluation that L is sequenced before.

This allows the reasoning about store-buffering to remain valid, while remaining implementable via OS-level primitives.

The problem with approach 2

While this approach meets criterion 1 (implementable via signals, membarrier(), FlushProcessWriterBuffers()), it fails criterion 2 (implementability via non-asymmetric fences).

It's not in general possible to takes as given (for sequentially consistent fences X and Y) "A is sequenced before X, X is before Y in the SC ordering, Y is sequenced before B" and conclude "A happens before B". Consider the following:

```
// Globals
int data{0};
atomic_int x{0};
atomic_bool P0_won_the_race{false};

// P0
data = 0x123456;
fence0();
if (x.load(relaxed) == 0) {
  P0_won_the_race.store(true, relaxed);
}

// P1
x.store(1, relaxed);
fence1();
if (P0_won_the_race.load(relaxed)) {
  assert(data == 0x123456);
}
```

Is the assertion permitted to fail? Yes, if `fence0()` and `fence1()` are `atomic_thread_fence(memory_order_seq_cst)` (the program has a data race on `data`, and therefore exhibits undefined behavior). The sequentially consistent portions of the fence rules only place restrictions on the coherence orders of affected object, not establish happens-before relationships between accesses (at least, not ones that an `memory_order_acq_rel` fence wouldn't, given the same observed values).

However, if `fence0()` and `fence1()` were an asymmetric fence pair (using approach 2), then we have that either the store to `data` strongly happens before P1's load, or that the store to `x` strongly happens before before P0's load (therefore ensuring that the load of data never happens). The asymmetric fence semantics are too strong to be implementable by `atomic_thread_fence` fences.

Interestingly, the assertion would be guaranteed to succeed if `data` were an atomic accessed only with relaxed loads and stores; this demonstrates that a data race may exist even if it can be shown that only a single store could satisfy a given load, if the variable raced upon had been atomic. This can appear only on architectures in which relaxed atomic loads and stores have stronger semantics than non-atomic loads and stores (or on implementations that optimize the latter more cleverly across fences).

Approach 3

We can fix the implementability problem of approach 2 by carving out a little bit more of its synchronization. In our proposed approach, we'll pick a definition of asymmetric fences that maps closely to the definition of plain fences. This has a couple of advantages:

- It satisfies the must-have criteria we listed above.
- It makes it easy to port code written with plain fences, while keeping most of the correctness arguments the same.
- It minimizes the mental overhead of the definitions; asymmetric fence semantics can be reasoned about using a small diff applied to plain fence semantics.
- Automated concurrency checkers can model asymmetric fences as plain fences and avoid introducing any new false positives.

We'll introduce two new events in the memory model: heavy and light asymmetric fences. Their semantics will closely follow those of plain fences, but omitting transitivity (to fix the issue from approach 1) and without giving any guarantees stronger than those provided by plain fences (to fix the issue from approach 2). In this section we'll outline a strategy for the definitions, by contrasting asymmetric fences with plain sequentially consistent ones. We'll consider non-sequentially-consistent fences and give formal wording subsequently.

Plain fences interact with one another in one of two ways: the synchronizes-with relation, and the "S" ordering (the total order of `memory_order_seq_cst` operations). The synchronizes-with relation is straightforward to translate; heavy asymmetric fences can synchronize with one another, or with plain fences. They *almost* synchronize with light fences, except that we have to strip out transitivity (to fix the correctness issue with approach 1).

The S ordering is trickier to translate; any two of its participants can be ordered with respect to one another, but this will not be the case with pairs of light fences. Instead, we'll pick our own way of deciding whether or not two asymmetric fences are ordered, but still borrow the consequences of relative ordering from plain fences.

For the first part (deciding how to order asymmetric fences), we introduce the asymmetrically-before relation, which connects light fences to heavy ones (and vice versa). It has two rules: for every two asymmetric fences X and Y, one of which is light and the other heavy,

- Either X is asymmetrically before Y, or Y is asymmetrically before X, but not both.
- If X happens before Y, then X is asymmetrically before Y.

(Note that this is implementable via plain sequentially consistent fences by letting the direction in asymmetrically-before be determined by the order in S).

For the second part (deciding the consequences of two asymmetric fences being ordered in a particular direction), recall that for plain sequentially consistent fences, we have a connection

between S and the modification order. It gives (assuming P0668) that, for A and B each operations on the same atomic object, and X and Y `memory_order_seq_cst` fences, the following holds: if X happens before A, A is coherence-ordered before B, and B happens before Y, then X is before Y in S. Our corresponding rule for asymmetric fences X and Y, one of which is light and the other heavy, is that, if A and B are operations on some atomic object, and X happens before A, A is coherence-ordered before B, and B happens before Y, then X is asymmetrically before Y.

This is the core of our proposed semantics. They could be strengthened somewhat, and still remain reasonable. For example: there's currently no guarantee that the union of asymmetrically before and happens before is acyclic (i.e if X is asymmetrically before Y and Y happens before Z, then there's no guarantee that X is asymmetrically before Z). However, absent an algorithm that requires such a guarantee, including it seems unnecessary.

Acquire and release asymmetric fences

In the last section, we did some hand-waving when claiming that asymmetric fences could borrow most of their synchronizes-with-lite semantics from plain sequentially consistent fences. Plain sequentially consistent fences only participate in synchronizes-with insofar as they are (definitionally) both acquire and release fences. In order to match their semantics, we need to translate the acquire and release definitions into an asymmetric form. We could therefore expose those features to users, with little increase in the complexity of the rules.

In this section, we'll argue that we ought to do this; there are algorithms that are most cleanly expressed using asymmetric fences taking memory orders. The final version of our proposed API (modulo naming) is then:

```
std::asymmetric_thread_fence_heavy(std::memory_order);
std::asymmetric_thread_fence_light(std::memory_order);
```

To see an example of why this might be advantageous, consider the following stripped-down implementation of `std::call_once`:

```
template <typename Func>
void call_once(once_flag& flag, Func func) {
  if (flag.initialized.load(std::memory_order_acquire)) {
    return;
  }
  lock_guard<mutex> lg(flag.mu);
  if (flag.initialized.load(std::memory_order_relaxed)) {
    return;
```

```
  }
  func();
  flag.initialized.store(true, std::memory_order_release);
}
```

This might be a candidate for asymmetric fences (weakening the acquire load to a relaxed load), if the already-initialized path is anticipated to be much more common than the uninitialized path. It could be rewritten as:

```
template <typename Func>
void call_once(once_flag& flag, Func func) {
  if (flag.initialized.load(std::memory_order_relaxed)) {
    asymmetric_thread_fence_light();
    return;
  }
  lock_guard<mutex> lg(flag.mu);
  if (flag.initialized.load(std::memory_order_relaxed)) {
    return;
  }
  func();
  asymmetric_thread_fence_heavy();
  flag.initialized.store(true, std::memory_order_relaxed);
}
```

On architectures where acquire loads have a cost (relative to relaxed loads), this speeds up the common path. But on architectures where acquire-loads are free (e.g. x86), it needlessly slows down the uninitialized path. There's no way to say "produce an asymmetric heavy fence, but only if release-acquire consistency has a cost otherwise". Without fences that take memory orders, the best implementation is something like the following:

```
template <typename Func>
void call_once(once_flag& flag, Func func) {
  if (flag.initialized.load(std::memory_order_relaxed)) {
    if (kRunningOnX86OrSparcTSO) { // Or some yet-to-be-invented arch?
                                   // Perhaps one the author doesn't know?
      std::atomic_thread_fence(std::memory_order_acquire);
    } else {
      asymmetric_thread_fence_light();
    }
    return;
  }
```

```
  lock_guard<mutex> lg(flag.mu);
  if (flag.initialized.load(std::memory_order_relaxed)) {
    return;
  }
  func();
  if (kRunningOnX86OrSparcTSO) {
    std::atomic_thread_fence(std::memory_order_release);
  } else {
    asymmetric_thread_fence_heavy();
  }
  flag.initialized.store(true, std::memory_order_relaxed);
}
```

Note that even this implies a false simplicity; `kRunningOnX86OrSparcTSO` may need to be determined at runtime. For example: the RISC-V memory model proposes a TSO-like ISA extension, relative to a weaker base specification. Portable code would be written for the weak memory model, but could query at runtime whether or not it's safe to assume TSO (in which case the heavy fence above can be omitted by branching on a global variable). Requiring the user to implement this logic (and get it right across all architecture/OS combinations) to ensure portably performant code is quite a lot to ask.

In our suggested semantics, asymmetric fences take a memory order. Light asymmetric fences would remain no-ops (except for their effects on compiler reordering). Release or acquire heavy fences would devolve to their non-asymmetric counterparts on architectures where its paired fence is free. We could then write the following:

```
template <typename Func>
void call_once(once_flag& flag, Func func) {
  if (flag.initialized.load(std::memory_order_relaxed)) {
    asymmetric_thread_fence_light(std::memory_order_acquire);
    return;
  }
  lock_guard<mutex> lg(flag.mu);
  if (flag.initialized.load(std::memory_order_relaxed)) {
    return;
  }
  func();
  asymmetric_thread_fence_heavy(std::memory_order_release);
  flag.initialized.store(true, std::memory_order_relaxed);
}
```

This would emit a heavy asymmetric fence down the initialization path on (e.g.) ARM, but not on (e.g.) x86, getting the desired implementation on both platforms. Note the close parallels to the non-asymmetric version.

Some commenters on a draft version of this paper did not find this example compelling, arguing that the high cost of the heavy fence made this implementation unreasonable. A more fleshed out implementation would alleviate some of these concerns. For example, the transition to the fenceless fast path could be done in bulk, for many different objects, and after some number of counter ticks. Even without such optimizations, the trade-offs may still make sense in a read-heavy workload.

Perhaps a more compelling example (albeit one too complicated to include in its entirety) is biased locking. A portably fence-free implementation requires three light asymmetric fences on the biased thread: one for the store-load barrier in the Dekker protocol, an acquire fence after the lock acquire, and a release fence before the lock release. This requires three heavy fences on the non-biased thread. On (e.g.) x86, two of these heavy fences are unnecessary; we very much don't want to force their inclusion. On (e.g.) Power, though, we very much do; if we omit them, and use plain fences on the common path, the lwsync instruction for the release fence alone can cost around 20 cycles. In many of the cases we're interested in, this can be greater than the cost of the critical section itself. Allowing the user to specify the required ordering precisely is the only way to avoid costs without introducing platform-specific reasoning into user code.

Standardese

Here we formalize the general strategy described above. Note that in the section giving release-acquire semantics, "A" and "B" are fences and "X" and "Y" are operations on an atomic object, while in the section giving sequentially consistent semantics, the roles are reversed (this matches an inconsistency in the current wording of plain fences, which should probably be fixed).

Asymmetric fences [atomics.fences.asymmetric]

This clause introduces *asymmetric fences*. Like (non-asymmetric) fences, asymmetric fences can have acquire semantics, release semantics or both, and may be sequentially consistent. Unlike non-asymmetric fences, they are either heavy or light.

If there exists an release asymmetric fence A, an acquire asymmetric fence B, with one of A and B light and the other heavy, and 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 it if were a release operation, then every evaluation that is sequenced before A strongly happens before every evaluation that is sequenced after B.

Sequentially consistent asymmetric fences additionally participate in the *asymmetrically before* relation, which obeys the following properties, where X and Y are sequentially consistent asymmetric fences, one of which is heavy and the other light

- Either X is asymmetrically before Y or Y is asymmetrically before X, but not both.
- If A and B are a pair of atomic operations on an atomic object, X happens before A, A is coherence-ordered before B, and B happens-before Y, then X is asymmetrically before Y.
- If X happens before Y, then X is asymmetrically before Y.

```
void asymmetric_thread_fence_heavy(memory_order order);
```

Effects: Performs `atomic_thread_fence(order);`. Additionally, depending on the value of `order`, this operation:

- Has no effects, if `order == memory_order::relaxed`
- Is an acquire heavy asymmetric fence, if `order == memory_order_acquire` or `order == memory_order_consume`
- Is a release heavy asymmetric fence, if `order == memory_order::release`
- Is both an acquire and a release heavy asymmetric fence, if `order == memory_order_acq_rel`
- Is a sequentially consistent acquire and release heavy asymmetric fence, if `order == memory_order_seq_cst`

```
void asymmetric_thread_fence_light(memory_order order);
```

Effects: Depending on the value of `order`, this operation:

- Has no effects, if `order == memory_order_relaxed`
- Is an acquire light asymmetric fence, if `order == memory_order_acquire` or `order == memory_order_consume`
- Is a release light asymmetric fence, if `order == memory_order_release`
- Is both an acquire and a release light asymmetric fence, if `order == memory_order_acq_rel`
- Is a sequentially consistent acquire and release light asymmetric fence, if `order == memory_order_seq_cst`

[Note: Delegating both heavy and light fence functions to an `atomic_thread_fence(order)` call with is a valid implementation.]

Possible objections

“What if my OS doesn’t have membarrier()?” / “What about freestanding implementations?”

Any implementation that actually improves common-path performance most likely requires functionality beyond traditional shared-memory, which is not universally available.

However, the semantics are defined so that falling back to `std::atomic_thread_fence(order)` for both asymmetric fences is a valid implementation strategy. If it’s unknown until runtime whether asymmetric fences are supported, branching on a global to decide whether or not to emit a fence will typically also be faster than actually emitting the fence.

Worth noting is that most (all?) atomics implementations already rely on at least some degree of runtime support for `std::atomic<LargeObject>`, so that this does not introduce a new need for a library dependency.

“Why isn’t RCU enough?”

Many data structures could use RCU (or, similarly, hazard pointers) to provide access to the magic fences; and implement their own versions as follows:

```
void asymmetric_thread_fence_light()
{
  rcu_reader rr;
}
```

```
void asymmetric_thread_fence_heavy()
{
  rcu_synchronize();
}
```

We don’t believe that this is a satisfactory alternative:

- It severely limits where heavy fences can be placed. They could not, for instance, be placed within a RCU read-side critical section, even though acquiring a biased lock within one is perfectly reasonable. Holding *any* mutex during the heavy fence is dangerous unless the caller knows that no thread has deferred a function object that will acquire the same mutex.
- It forces unnecessary efficiency tradeoffs. In at least one reasonable implementation of RCU (i.e. folly’s), `rcu_synchronize()` requires two heavy asymmetric fences, doubling the cost of the heavy fence for users who don’t need `rcu_synchronize()` semantics.

“What about implementation scalability?”

Current IPI-based implementations of the heavy fence can require time proportional to the number of cores. It’s reasonable to worry that what works for low-processor-count systems might not scale in the future. However, this need not be the case; the linear cost is an artifact of current implementation strategies. A more advanced approach could use a tree-structured broadcast algorithm to achieve heavy fence latencies that are logarithmic in the number of cores.

Paul McKenney writes:
It is eminently susceptible to divide-and-conquer techniques, so in theory it should scale extremely well. In practice, the current Linux-kernel sys_membarrier() implementation does not yet make use of these techniques because we haven't yet seen the need, so we haven't bothered. But if someone were using sys_membarrier() to implement RCU or hazard pointers on a system with (say) 64K CPUs, I am reasonably sure that divide-and-conquer would be needed.
Another way to look at this that might (or might not help) is as a broadcast operation (the IPIs in expedited sys_membarrier()) followed by a reduce operation (waiting for the IPIs to complete).

“What about MEMBARRIER_CMD_[...]_EXPEDITED?”

The Linux `membarrier()` system call takes a `flags` argument which allows specifying whether or not the call is expedited (which, in practice, determines whether or not the effects are achieved by inter-processor interrupts or by waiting for a scheduling quantum to expire on all cores; the former gives lower latency for heavy fences, the latter higher throughput on other cores).

We do not suggest exposing this to end users; it’s too non-portable, and is better left as an extension if implementations which to expose it (or as an options argument added to the interface later on, if necessary). The author has never seen a “real” use case in which the non-expedited variant is preferable, and suggests that the expedited version be used by default, when available.

“membarrier() doesn’t take a memory order; why should we?”

The OS-provided implementation techniques don’t vary their ordering semantics for heavy fences; some reviewers found it odd that our proposed API does.

The reason we include memory ordering arguments is not to affect the semantics of the `membarrier()` call; it’s to allow us to omit it entirely where the architecture provides the necessary effects costlessly. Release-acquire semantics are “free” on some architectures and have a cost on others. Taking the memory order in the fence lets these constraints be

expressed without making users encode architecture specific reasoning about whether or not to emit a heavy fence or a no-op (except for compiler reordering) `atomic_thread_fence`.

Naming concerns

Repeating “heavy asymmetric fence” and “light asymmetric fence” (and thereby leaving an anadorned “fence” somewhat vague-sounding) is quite the mouthful. We could pick pithier, more distinctive names; perhaps hfence and lfence, or heavy afence and light afence. Or remove the “asymmetric”, and simply say “heavy fence “ and “light fence”, leaving “asymmetric” implied.

These are concerns only for standards readers; the user-visible portions of the API are fairly unambiguous.

The API-visible names of those fences, on the other hand, are a promising avenue for many happy hours of bikeshedding. Perhaps they should (like `atomic_thread_fence` and `atomic_signal_fence`) only vary the middle word of the name, and be called `atomic_heavy_fence` and `atomic_light_fence`. Or perhaps the pattern should be `atomic_[participant_in_synchronization]_[type_of_synchronization]`, and they should be named `atomic_thread_heavy_fence` and `atomic_thread_light_fence`. Most such choices are reasonable.

Links

Sampling of uses

Hotspot’s os::serialize_thread_states():
https://github.com/JetBrains/jdk8u_hotspot/blob/435f973f98771edfa2126d5e6b6dea9bbf272e86/src/share/vm/runtime/os.cpp#L1294

Grand Central Dispatch’s dispatch_once():
https://opensource.apple.com/source/libdispatch/libdispatch-339.90.1/src/once.c

Various folly uses:
https://github.com/facebook/folly/search?q=asymmetricLightBarrier
https://github.com/facebook/folly/search?q=asymmetricHeavyBarrier
https://github.com/facebook/folly/search?q=TLRefCount
https://github.com/facebook/folly/search?q=ThreadCachedInts

.NET CLR:
https://github.com/dotnet/coreclr/search?q=FlushProcessWriteBuffers&unscoped_q=FlushProcessWriteBuffers

liburcu:
https://github.com/urcu/userspace-rcu/blob/3745305bf09e7825e75ee5b5490347ee67c6efdd/src/urcu.c#L170

QEMU:
https://github.com/qemu/qemu/blob/915d34c5f99b0ab91517c69f54272bfdb6ca2b32/util/sys_membarrier.c#L24
https://github.com/qemu/qemu/search?q=smp_mb_global

The Objective-C runtime's _collecting_in_critical():
https://opensource.apple.com/source/objc4/objc4-646/runtime/objc-cache.mm

Other discussions

"Biased Locking in Hotspot" - Dave Dice:
https://blogs.oracle.com/dave/biased-locking-in-hotspot

"How to Implement Unnecessary Mutexes" - Mike Burrows:
https://rd.springer.com/chapter/10.1007/0-387-21821-1_7

"Secrets of dispatch_once" - Mike Ash:
https://www.mikeash.com/pyblog/friday-qa-2014-06-06-secrets-of-dispatch_once.html

"QPI Quiesence" - Dave Dice:
https://blogs.oracle.com/dave/qpi-quiescence