Document number: P1202R1
Date: 2018-01-20 (pre-Kona)
Reply-to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: SG1

P1202R1: 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.

History

In San Diego, SG1 took the following poll, in the discussion of the R0 version of this paper:
We are interested in this direction for a TS; we want to do further wording review
SF F N A SA
7 3 3 0 0
This iteration is thematically similar, but emphasizes the wording-relevant portions more heavily.

One of the questions from the discussion was (quoting from the minutes): "In what sense are the not sequentially consistent fences sequentially consistent?"; the only real answer was that you use them in the places you would otherwise use memory_order_seq_cst fences (e.g. to order stores with subsequent loads). The semantics here are strengthened slightly relative to R0, to give a more satisfying answer; sequentially consistent asymmetric fences are first-class citizens of the SC ordering. However, no consequences can be derived from a light asymmetric fence's inclusion in the ordering, except when viewed relative to a heavy fence. The implications (i.e. restrictions imposed on coherence ordering) of the light fence with respect to the heavy fence is then the same as if all the fences in questions were atomic_thread_fence(memory_order_seq_cst). (There are still no extra inferences one can draw between two light fences, even if they're sandwiched between different pairs of heavy fences).

This wasn't polled explicitly, but SG1 seemed of the opinion that the asymmetric fence functions should be noexcept (on the basis that failures would be unrecoverable). The suggested wording has been updated accordingly.

A useful development between R0 and R1 is a discussion spread across the glibc and Linux mailing lists, on the formal semantics of sys_membarrier (Google “[PATCH] Linux: Implement membarrier function”). Several Linux folks seemed to settle on some RCU-inspired ordering guarantees for membarrier(), including patches to the herd formalization of the Linux kernel memory model implementing them. They aren’t official, and I don’t have any theorem-prover verified equivalences, but the fact that the semantics there are at least as strong as the semantics proposed here on a variety of litmus tests increases my confidence in these.

Recap of Argument for Inclusion in a TS

Some OSs provide a system call (or some other abstraction that can use system-specific features to break past the ordinary limitations of shared memory) that has the effect of ensuring that a memory barrier gets inserted asynchronously into the instruction stream of other threads. In situations that synchronize between two paths, one very common and one very uncommon, the high cost of the OS abstraction on the uncommon path is paid for by the benefits incurred on the common one (which can avoid the barrier entirely; it will get inserted if necessary, after a request from the uncommon path). These techniques are increasingly common, used in language runtimes (e.g. Hotspot), concurrency libraries (e.g. liburcu), and general purpose abstractions (e.g. several in folly). When those OS calls aren’t available, programmers sometimes work around their absence, with techniques ranging from the ugly (signals, explicit quiescence periods) to the profane (mprotect-triggered TLB shootdowns, misaligned atomic RMWs). Judicious use of the OS primitives can result in 10x speedups (or more) down fast paths.

We propose exposing these OS primitives, through an `asymmetric_thread_fence_[light|heavy]()` API that remains implementable via fallback to plain fences, for portability. Since some architectures have costless acquire and release fences while others do not, it’s hard for users to express “insert a plain fence, but only if it’s free; otherwise use a asymmetric fence” (this comes up, for example, in implementing biased locking portably and efficiently across x86 and Power). To make this easy to get “right” portably, we give the asymmetric fences a memory order parameter. Acquire and release asymmetric fences can cheaply devolve to the non-asymmetric equivalent on TSO architectures, while using truly asymmetric functionality on weaker ones. Because some implementations target platforms that don’t have underlying OS support (or indeed, even an underlying OS), we’ll need to be careful to make sure that the semantics we pick remain correct if every light and heavy fence is replaced by its non-asymmetric counterpart.

This summary omits many details; the R0 version has more in-depth arguments for the overall shape of the design.

Some synchronization properties of asymmetric techniques

Relationship between light and heavy fences

Here, I'll argue (informally) that all reasonable asymmetric techniques satisfy the following property:

For every light fence L and heavy fence H, one of the following holds:

- Everything sequenced before L strongly happens before everything sequenced after H
- Everything sequenced before H strongly happens before everything sequenced after L

Implementations boil down to a heavy fence having a synchronization point with some particular thread: the point in the target thread's instruction stream at which the polling or interruption occurs.

Correctness for polling-based techniques

Here, the compiler inserts explicit periodic checks to see if other threads have a pending unacknowledged heavy fence. (These checks need to be periodic rather than only occurring at light fences in order to ensure progress for heavy-fencing threads even if other threads never perform light fences). In such cases, the polling strategy can check to see if there is a pending request, fence if so, and send a response. For these cases, the result is trivial; there is release-acquire synchronization between the heavy fence request and the target thread's receipt of it, and release-acquire synchronization between the target thread's response and the heavy-fencing thread's receipt of the response.

Correctness for interrupt-based techniques

Here, the heavy-fencing thread sends some sort of signal (or inter-processor interrupt, etc.) to the other threads; a fence is inserted at some point in their (possibly compiler-reordered) instruction stream. A light fence is just a compiler barrier.

The interrupt occurs at some point in the target thread's instruction stream; it is either before the compiler barrier, at the point of the compiler barrier, or after the compiler barrier. The first bullet point holds in the first two cases (assuming a sufficiently synchronizing interrupt handler), and the second bullet point holds in the second two cases.

The interrupt might be received in the middle of an atomic operation (since even loads and stores may be implemented as "load; fence" or "store; fence", or might be a RMW implemented

using an LL/SC primitive). In practice, the interrupt receipt pathways include equivalent fences anyways, and implementations need to be resilient to spurious LL/SC failures for unrelated reasons. The existing memory model also breaks if signal handlers are allowed to execute without a degree of implied fencing (see the reflector thread “MM implementability in the presence of signals”). We don’t need to reason about interrupt receipt within non-atomic operations, since a program that can tell the difference is racy to begin with.

A motivating Litmus test

To see why we picked the above criterion, rather than the similar but simpler “either H synchronizes with L or L synchronizes with H”, consider the following litmus test (which will show that the simpler version is not correct).
Consider the following program:

`// 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).

In a polling implementation, the result seems more obvious; the heavy fence can have its requests acknowledged before the first instruction of T2, and after the last instruction of T0.

Strengthening the guarantees

In implementing the library, we can add a plain sequentially consistent fence before we begin invoking the underlying heavy fence implementation. This has two consequences, one obvious and one non-obvious:

- We can use the sequentially consistent fence as the whole heavy fence's position in the SC ordering, and satisfy all the guarantees required with respect to other (non-light) SC operations.
- For the purposes of constraining the values that can be obtained for loads of atomic variables, we can assume that the synchronization points of two heavy fences with respect to any given target thread occur in the same order as that of their leading fences in the SC order; if H1's initial SC fence precedes H2's initial SC fence, then for each thread T we can pretend that H1's interruption or poll receipt occurred before H2's in T's execution, even though they could have occurred out of order in reality.

The justification for the second bullet point is that, in moving the later synchronization point earlier in T's execution, the only thing we're doing is adding constraints on the values that could be obtained by loads between the old and new positions. But those values are already constrained by the SC ordering; H1's SC fence precedes H2's SC fence, which strongly happens before everything after the synchronization point's fence in the ordering (from the synchronization property above). H1's thread has not read any new (user-visible) values in between its fence and its synchronization point with T, so the constraints the movement would add were already required to begin with.

This argument is what justifies this paper's strengthening relative to R0's.

Wording

Light fences don't establish ordering constraints with non-asymmetric fences. The existing wording just refers to "fences"; we need to decide if an "asymmetric fence" is a type of "fence". There are several options:

- Decide that an asymmetric fence is not a type of fence. In this case, leaving the name "fence" in them is a little confusing; we could rename them to "heavy afence" and "light afence", or "hfence" and "lfence", or something similar.
- Decide that heavy asymmetric fences *are* plain fences, but that light ones are not. This introduces a bit of an ugly asymmetry (and we have the same naming problem as above for what to call a light fence).
- Decide that both heavy and light fences are types of fence, but specifically exclude light fences from the requirements of [atomics.fences].
- Change the existing usage of "fence" to (something like) "plain fence". This lets us introduction a "fence" category including both "plain fences" and "asymmetric fences", whose intersection is "heavy asymmetric fences". This makes the asymmetric fence wording trickier to read in TS form, and makes the term "fence" vague if it's unclear if the context is "IS + TS" or "just IS".

The wording below takes the first approach, calling the two asymmetric fence types "lfence" and "hfence".

Legalese

Asymmetric fences [atomics.fences.asymmetric]

This section introduces synchronization primitives called *hfences* and *lfences*. Like fences, hfences and lfences can have acquire semantics, release semantics, or both, and may be sequentially consistent (in which case they are included in the total order S on `memory_order::seq_cst` operations).

If there are evaluations A and B, 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 if it were a release operation, and one of the following hold:

- A is a release lfence and B is an acquire hfence; or
- A is a release hfence and A is an acquire lfence

then any evaluation sequenced before A strongly happens before any evaluation that B is sequenced before.

If there are evaluations A and B, 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 if it were a release operation, and one of the following hold:

- A is a release fence and B is an acquire hfence; or
- A is a release hfence and B is an acquire fence; or
- A is a release hfence and B is an acquire hfence

then A synchronizes with B.

For every pair of atomic operations A and B on an object M, where A is coherence-ordered before B, the total order S on all memory_order::seq_cst operations obeys the following properties:

- if A is a memory_order::seq_cst operation and B happens before a memory_order::seq_cst hfence Y, then A precedes Y in S; and
- if a memory_order::seq_cst hfence X happens before A and B is a memory_order::seq_cst operation, then X precedes B in S; and
- if a memory_order::seq_cst lfence X happens before A and B happens before a memory_order::seq_cst hfence Y, then X precedes Y in S; and
- if a memory_order::seq_cst hfence X happens before A and B happens before a memory_order::seq_cst lfence Y, then X precedes Y in S; and
- if a memory_order::seq_cst hfence X happens before A and B happens before a memory_order::seq_cst hfence Y, then X precedes Y in S.

[Note: the constraints implied by a fence and an hfence, or by two hfences, are the same as would be implied by replacing hfences by fences with the same `memory_order`.]

[Note: The requirements that the strongly happens before relation places on S are not relaxed for hfences or lfences.]

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

Effects: Depending on the value of `order`, this operation:
- Has no effects, if `order == memory_order::relaxed`
- Is an acquire hfence, if `order == memory_order_acquire` or `order == memory_order_consume`
- Is a release hfence, if `order == memory_order::release`
- Is both an acquire and a release hfence, if `order == memory_order_acq_rel`
- Is a sequentially consistent acquire and release hfence, if `order == memory_order_seq_cst`

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

Effects: Depending on the value of `order`, this operation:
- Has no effects, if `order == memory_order_relaxed`
- Is an acquire lfence, if `order == memory_order_acquire` or `order == memory_order_consume`
- Is a release lfence, if `order == memory_order_release`
- Is both an acquire and a release lfence, if `order == memory_order_acq_rel`
- Is a sequentially consistent acquire and release lfence, 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.]

Correctness

We're interested in two implementations: those that use truly asymmetric techniques, and those that fall back to plain fences (i.e. replacing both the light and heavy fences with their non-asymmetric counterparts). The former is needed for the performance improvement we're seeking, and the latter is needed for portability.

(In this section, arguments are relative to the wording after the merge of P0668).

Plain fence implementation

The acquire/release guarantees between heavy and heavy, or heavy and plain fences are the same as for plain fences. Moreover, a release fence synchronizes with an acquire fence, and therefore simply happens before it, and so evaluations sequenced before the release fence strongly happen before those sequenced after the acquire fence, which is the guarantee that heavy/light fences provide.

The SC operations fall out of the existing definitions easily; in all cases, we've just taken existing wording and applied it to either fence/hfence pairs or lfence/hfence pairs.

Asymmetric implementations

Here, "light fence" and "heavy fence" refer to the true asymmetric implementations, while lfence and hfence refer to the actions in the memory model whose semantics we want to justify in terms of those underlying implementations.

Consider a light (resp. heavy) fence A and a heavy (resp. light) fence B, and suppose that there are 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 blah blah blah release sequences; the argument below will work even with modification order).

Either everything sequenced before A strongly happens before everything B is sequenced before, or everything sequenced before B strongly happens before everything A is sequenced before (from the "Relationship Between Light and Heavy Fences" section). If the latter, then Y strongly happens before X, and therefore could not have obtained its value from X or any subsequent operation in M's modification order. So we must be working in the former case, which is exactly the postcondition we want to establish.

The SC postconditions are trickier to justify; we'll use the trick from the "Strengthening the guarantees" subsection, and note that we can place an SC fence before the heavy side of the asymmetric implementation (or at least, imagine that we have; implementations can of course do whatever they want). This ensures that heavy fences interact with one another and with plain fences just as a plain fence in the same spot would. This in turn implies all the SC postconditions that do not involve lfences.

In the same subsection, we argued that we can assume that the synchronization point of heavy fences in every light thread's execution stream occurred in the same order as the order of their preceding plain SC fences. So, for each light fence L on a thread T, we can assume there is some last (in the SC ordering) heavy fence that has its synchronization point with T before L, and the next heavy fence (again, in the SC ordering) has its synchronization point with T after L. We can place the lfence associated with a given light fence at any point in the SC ordering

between those two heavy fences; the location of an lfence in the SC ordering is not otherwise observable by the postconditions we provide.