Document number: P1202R2
Date: 2019-06-16 (pre-Cologne)
Reply-to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: LEWG

P1202R2: 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.

The proposed ship vehicle is Concurrency TS 2.

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

In Kona, SG1 did such a wording review, and took the following poll on the R1 wording:
Forward to LEWG for consideration in concurrency TS v2
SF F N A SA
4 8 4 0 0

(Correspondingly, I recommend the R0 version of this paper for an overview of the technique and its use cases, and the R1 version for an argument of the correctness of the following wording).

This includes just the wording from the R1 version, with minor typos fixed.

Wording

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 B 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 an acquire hfence 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 an acquire lfence 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.]

Typos fixed

The changes from the R1 wording that SG1 reviewed are the following:

- “A is a release hfence and A is an acquire lfence” -> “A is a release hfence and B is an acquire lfence”
- “acquire and release [h|l]fence” -> “acquire [h|l]fence and a release [h|l]fence”.

Commentary

Much of the wording here echoes wording from corresponding sections of the plain fence semantics. Arguably, both could be improved in places; there is value though, in keeping the wording similar.

Here is a list of questions that SG1 hopes the TS process may provide useful information on (roughly taken from the minutes of 2019 Kona discussion, but not verbatim quotes):

- Should the asymmetric fence functions be noexcept? SG1 seemed to lean (non-polled) towards yes. It was pointed out that, if we make them noexcept for the TS versions, we'll find out if they should have thrown; if we don't make them noexcept in the TS but should have, we'll never get that feedback.
- Are there any bugs that will be exposed by formal modelling (or practice)?
- Is this the right interface? Does it support an efficient implementation on a broad range of vendor platforms? We are particularly curious about this question for GPUs.
- How often is this feature used? (If it's never practically useful, perhaps it is not worth standardizing. If it's mostly an attractive nuisance, with users reaching for it when they should be using plain fences, then perhaps it's actually harmful).
- Will this interact in surprising ways with other features? Coroutines, executors, and freestanding were brought up explicitly.
- Are there implementations where users want an explicit options argument? (E.g. on Linux, sys_membarrier can take a variety of flags).