Document number: P1202R3
Date: 2021-05-27
Reply-to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: LWG

P1202R3: 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

In Prague, LEWG voted (unanimous consent) to forward to LWG for Concurrency TS 2.

(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).

R2 included the R1 wording with minor typos fixed.

R3 (this version) is the R2 wording, with changes applied during LWG pre-review; adding section and header names, including a synopsis, placing functions in the proper namespace.

Wording

31.2 Header <atomic> synopsis [atomics.syn]

Add the following declarations to the synopsis of the header `<atomic>`:

```
namespace std::experimental::inline concurrency_v2 {
  // ?.2.1 asymmetric_thread_fence_heavy
  void asymmetric_thread_fence_heavy(memory_order order) noexcept;
  // ?.2.2 asymmetric_thread_fence_light
  void asymmetric_thread_fence_light(memory_order order) noexcept;
}
```

31.4 Order and consistency [atomics.order]

In subclause 31.4 [atomics.order], strike the word “four” in the phrase “the following four conditions are required to be satisfied by S:” and add the following two bullets to the list:

- if a `memory_order::seq_cst` lightweight-fence X happens before A and B happens before a `memory_order::seq_cst` heavyweight-fence Y, then X precedes Y in S; and
- if a `memory_order::seq_cst` heavyweight-fence X happens before A and B happens before a `memory_order::seq_cst` lightweight-fence Y, then X precedes Y in S.

31.? Asymmetric fences [atomics.fences.asym]

This section introduces synchronization primitives called *heavyweight-fences* and *lightweight-fences*. Like fences, heavyweight-fences and lightweight-fences can have acquire semantics, release semantics, or both, and can be sequentially consistent (in which case they are included in the total order S on `memory_order::seq_cst` operations). A heavyweight-fence has all the synchronization effects of a fence as specified in 31.11 [atomic.fences]. [Note: Heavyweight-fences and lightweight-fences are distinct from fences. -- end note]

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

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

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

1. Effects: Depending on the value of `order`, this operation:
   - has no effects, if `order == memory_order::relaxed`;

- is an acquire heavyweight-fence, if `order == memory_order::acquire` or `order == memory_order::consume`;
- is a release heavyweight-fence, if `order == memory_order::release`;
- is both an acquire heavyweight-fence and a release heavyweight-fence, if `order == memory_order::acq_rel`;
- is a sequentially consistent acquire and release heavyweight-fence, if `order == memory_order::seq_cst`.

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

1. Effects: Depending on the value of `order`, this operation:
   - has no effects, if `order == memory_order::relaxed`;
   - is an acquire lightweight-fence, if `order == memory_order::acquire` or `order == memory_order::consume`;
   - is a release lightweight-fence, if `order == memory_order::release`;
   - is both an acquire lightweight-fence and a release lightweight-fence, if `order == memory_order::acq_rel`;
   - is a sequentially consistent acquire and release lightweight-fence, if `order == memory_order::seq_cst`.

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

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).