What’s up with this new memory_order_consume memory order?

devblogs.microsoft.com/oldnewthing/20230427-00

April 27, 2023

Raymond Chen

C++20 introduces a new atomic memory order: `std::memory_order::consume`, more commonly known as `std::memory_order_consume`, What is this guy?

The `consume` memory order is a weaker form of `acquire`. Whereas `acquire` prevents *all* future memory accesses from being ordered ahead of the load, the `consume` order only prevents *dependent* future memory accesses from being reorder ahead of the load.

In all the examples, let’s assume global variables declared and initialized as

```
int v1 = 1;
int v2 = 2;
std::atomic<int*> p{ &v1 };
```

Okay, let’s do some consuming.

```
auto sample_consume()
{
    auto q = p.load(std::memory_order_consume);
    return *q + v2;
}
```

The compiler is required to read the value from `p` into `q`, and any future calculations depending on that value must occur after the load.

This reordering is allowed:

```
auto sample_consume_allowed()
{
    auto prefetch2 = v2;
    auto q = p.load(std::memory_order_consume);
    return *q + prefetch2;
}
```

The value of `v2` is not dependent on what was loaded from `p`. Therefore, the compiler and processor are permitted to advance the fetch of `v2` ahead of the load of `p`. Note that an `acquire` load of `p` would have prohibited this reordering, since acquire loads block *all* future memory access, even if unrelated to the value being acquired.

However, this reordering of the above code is not allowed:

```
auto sample_consume_disallowed()
{
    auto speculate1 = v1;
    auto q = p.load(std::memory_order_consume);
    if (q == &v1) return speculate1 + v2;
    return *q + v2;
}
```

This speculation lets the code hide the memory latency of accessing `v1` behind the load of `p`, and a compiler might choose to take advantage of this based on profiling feedback, and a processor might do it unilaterally because processors like to do speculative things nowadays. This would be allowed if the load from `p` were `relaxed`.

However, the `consume` memory order prohibits this transformation: The value loaded from `p` is dereferenced, and that dereference operation is dependent upon the value that was loaded, so the `consume` memory order requires that the dereference occur after the load.

Here's a table, because people like tables.

Ordering	Relaxed	Consume	Acquire
Load `v2` before `p`	Allowed	Allowed	Prohibited
Dereference `p` before load	Allowed	Prohibited	Prohibited

The `consume` memory order is not used much. Atomic variables are typically tied to other variables in ways that don't show up in expression dependency graphs, such as for use as mutual exclusion locks. The `acquire` memory order is much more commonly used than `consume`.