How can I do the opposite of compare_exchange and exchange if the value is different?

devblogs.microsoft.com/oldnewthing/20221209-00

December 9, 2022

Raymond Chen

The C++ language comes with `std::atomic::compare_exchange_strong()` to compare the current value of an atomic variable to a specific provided value and exchange if they are equal. But how do you do the reverse and exchange if they are *unequal*?

Just exchange it. An exchange with itself has no effect on the value.

```
std::atomic<int> var;

// Atomically "var" back to 0, and if it was nonzero,
// then do some cleanup.
bool try_reset()
{
    if (var.exchange(0) != 0) { cleanup(); }
}
```

If the previous value was nonzero, then the exchange will set it to zero and return the previous nonzero value. We then detect this nonzero value and perform the cleanup.

If the previous value was zero, then the exchange will set it to zero (having no net effect on the value since it was already zero) and return the previous zero value. We detect this zero value and bypass the cleanup.

Note that this will generate a write to memory due to the exchange, so even though the effect is the same, you do suffer potential cache contention issues. To avoid that, you can early-out the case where the value is already zero.

```
// Atomically "var" back to 0, and if it was nonzero,
// then do some cleanup.
bool try_reset()
{
    if (var.load() != 0) {
        if (var.exchange(0) != 0) { cleanup(); }
    }
}
```

This avoids a spurious write to `var` in the case where the value is already zero, but it costs you an extra barrier if the value was nonzero, in order to preserve sequential consistency.

Depending on your use case, you may be able to weaken the ordering. Maybe an acquire on the load and a release on the exchange are sufficient. I'll let you decide what your requirements are.

Now, maybe the value you want to exchange in is different from the "nop" value. In that case, you'll have to split it into two steps, one to check whether the exchange is necessary, and the other to perform it conditionally. We saw how to do this some time ago: The lock/commit/(try again) pattern is a case of the extensible interlocked operation pattern. You perform a provisional computation based on the current value, and then use a "compare-exchange" to try to commit it. If it fails, then go back and try again.

```
bool exchange_unless(int value, int bad_value)
{
    int old_value = var.load(std::memory_order_acquire);
    do {
        if (old_value == bad_value) return false;
    while (!var.compare_exchange_weak(value, old_value,
                std::memory_order_release));
    return true;
}
```

Bonus chatter: There is currently no way of expressing it in C++, but this pattern fits very closely with the the load-locked/store-conditional pattern of most RISC processors.

```
exchange_unless:
    ; assume r0 = value, r1 = bad_value

    memory_barrier
    load_address r2 = [var]
retry:
    load_locked r3 = [r2]
    conditional_branch r3 == r1, failed
    store_conditional [r2] = r0, retry

    memory_barrier
    move r0 = #1
    return

failed:
    move r0 = #0
    return
```

The details of the instructions vary from processor to processor, but the overall shape looks the same.

Raymond Chen

Follow