Lock-free algorithms: The try/commit/(try again) pattern

devblogs.microsoft.com/oldnewthing/20110412-00

April 12, 2011

Raymond Chen

The singleton constructor pattern and the `InterlockedMultiply` example we saw some time ago are really special cases of the more general pattern which I'll call try/commit/(try again). I don't know if this pattern has a real name, but that's what I'm calling it for today.

The general form of this pattern goes like this:

```
for (;;) {
 // capture the initial value of a shared variable we want to update
 originalValue = sharedVariable;
 ... capture other values we need to perform the operation ...
 ... these values must be indepedent of sharedVariable ...
 newValue = ... calculate the desired new result
                based on originalValue and other captured values ...
 // Xxx can be Acquire, Release, or null
 if (InterlockedCompareExchangeXxx(
            &sharedVariable,
            newValue, oldValue) == oldValue) {
  break; // update was successful
 }
 ... clean up newValue ...
} // loop back and try again
```

We calculate the desired new value based on the initial value, combining it with other values that vary depending on the operation you want to perform, and then use an `InterlockedCompareExchange` to update the shared value, *provided the variable hasn't changed from its initial value*. If the value did change, then that means another thread raced against us and updated the value before we could; in that case, we go back and try it again. Maybe the next time through we won't collide against somebody.

Note that the try/commit/try again pattern is unfair. There's no assurance that the thread that has been trying to update the value for the longest time will win the next race. (This is a property common to most lock-free algorithms.)

The `InterlockedMultiply` function follows this pattern very closely: The other value required to perform the operation is simply the multiplier, which is a parameter to the function and therefore is independent of the shared variable. The new value is simply the product, and if we are unable to update the shared value (because somebody else modified it), we just start over.

A variation of try/commit/try again is try/commit/abandon. In this pattern, there is no loop. If the exchange fails, you just give up and return a failure code. The function `TryEnterCriticalSection` uses the try/commit/abandon pattern. (It also uses the Acquire version of `InterlockedCompareExchange` for reasons which should be obvious.)

Our singleton pattern is another special case of try/commit/try again where the "try again" is optimized out because we know what the result of "try again" is going to be, so we don't actually have to do it. In the singleton pattern case, the `InterlockedCompareExchange` is a Release because the new value depends on other memory locations.

Normally, the shared variable is an integer rather than a pointer, because a pointer is subject to the ABA problem if you incorporate the pointed-to data into your calculations. We get away with it in the singleton pattern case because the value change is unidirectional: It goes from `NULL` to *something*, and once it's *something* it never changes again. If the value of the shared variable can change in more general ways, then you have to be more careful if you use a pointer as the shared variable. (The most common solution is to make the shared variable not just a pointer but a pointer plus a counter which increments at each operation.)

Here's another use of the try/commit/try again pattern, using a change counter as the shared variable. First, two helper functions:

```
LONG InterlockedReadAcquire(__in LONG *pl, __in LONG lUnlikely)
{
  return InterlockedCompareExchangeAcquire(pl, lUnlikely, lUnlikely);
}
LONG InterlockedReadRelease(__in LONG *pl, __in LONG lUnlikely)
{
  return InterlockedCompareExchangeRelease(pl, lUnlikely, lUnlikely);
}
```

Although direct reads and writes of properly aligned `LONG`s are atomic, the operations are not synchronized and impose no memory ordering semantics. To read a value with specific semantics, I pull a sneaky trick: I perform an `InterlockedCompareExchange` with the desired memory ordering semantics, passing the same value as the comparand and the exchange; therefore, the operation, even if successful, has no computational effect.

```
if (*pl == lUnlikely) *pl = lUnlikely;
```

To avoid dirtying the cache line, I use an unlikely value as the comparand/exchange, so most of the time, the comparison fails and no memory is written. (This trick doesn't help on all architectures, but it doesn't hurt.)

Okay, back to the change counter example:

```
LONG g_lColorChange;
...
case WM_SYSCOLORCHANGE:
 InterlockedIncrement(&g_lColorChange);
 ...
int CalculateSomethingAboutSystemColors()
{
 LONG lColorChangeStart;
 do {
  lColorChangeStart = InterlockedReadAcquire(&g_lColorChange, -1);
  COLORREF clrWindow = GetSysColor(COLOR_WINDOW);
  COLORREF clrHighlight = GetSysColor(COLOR_HIGHLIGHT);
  ... other computations involving GetSysColor(...)
 } while (InterlockedReadRelease(
                       &g_lColorChange, -1) != lColorChangeStart);
 return iResult;
}
```

We capture the color change counter and then begin doing our calculations. We capture the value with acquire semantics so that we get the value before we start reading the system colors. When we're done, we compare the value of the change counter against the value we captured. If it's different, then that means that the colors changed while we were doing our calculations, so our calculations are all messed up. In that case, we go back and try it again.

This technique does assume that you won't get into a situation where one thread manages to increment the change counter 4 billion times before the other thread manages to run. This is not a problem in practice. For example, in this case, it's reasonable to assume that nobody is going to change their system colors 4 billion times within a single thread quantum.

Next time, I'll show a different variation on try/commit/abandon which might be suitable for simple caches.

Exercise: Criticize the following: "I noticed that there is no interlocked read operation, but there is `InterlockedOr`, so my plan is to perform an interlocked read by or'ing with zero."

Raymond Chen

Follow