Interlocked operations don’t solve everything

devblogs.microsoft.com/oldnewthing/20040915-00

September 15, 2004

Raymond Chen

Interlocked operations are a high-performance way of updating DWORD-sized or pointer-sized values in an atomic manner. Note, however, that this doesn’t mean that you can avoid the critical section.

For example, suppose you have a critical section that protects a variable, and in some other part of the code, you want to update the variable atomically. “Well,” you say, “this is a simple imcrement, so I can skip the critical section and just do a direct InterlockedIncrement. Woo-hoo, I avoided the critical section bottleneck.”

Well, except that the purpose of that critical section was to ensure that nobody changed the value of the variable while the protected section of code was running. You just ran in and changed the value behind that code’s back.

Conversely, some people suggested emulating complex interlocked operations by having a critical section whose job it was to protect the variable. For example, you might have an InterlockedMultiply that goes like this:

```
// Wrong!
LONG InterlockedMultiply(volatile LONG *plMultiplicand, LONG lMultiplier)
{
  EnterCriticalSection(&SomeCriticalSection);
  LONG lResult = *plMultiplicand *= lMultiplier;
  LeaveCriticalSection(&SomeCriticalSection);
  return lResult;
}
```

While this code does protect against two threads performing an InterlockedMultiply against the same variable simultaneously, it fails to protect against other code performing a simple atomic write to the variable. Consider the following:

```
int x = 2;
Thread1()
{
  InterlockedIncrement(&x);
}


Thread2()
{
  InterlockedMultiply(&x, 5);
}
```

If the InterlockedMultiply were truly interlocked, the only valid results would be x=15 (if the interlocked increment beat the interlocked multiply) or x=11 (if the interlocked multiply beat the interlocked increment). But since it isn't truly interlocked, you can get other weird values:

Thread 1	Thread 2
x = 2 at start	
	InterlockedMultiply(&x, 5)
	EnterCriticalSection
	load x (loads 2)
InterlockedIncrement(&x); x is now 3	
	multiply by 5 (result: 10)
	store x (stores 10)
	LeaveCriticalSection
x = 10 at end	

Oh no, our interlocked multiply isn't very interlocked after all! How can we fix it?

If the operation you want to perform is a function solely of the starting numerical value and the other function parameters (with no dependencies on any other memory locations), you can write your own interlocked-style operation with the help of InterlockedCompareExchange.

```
LONG InterlockedMultiply(volatile LONG *plMultiplicand, LONG lMultiplier)
{
  LONG lOriginal, lResult;
  do {
    lOriginal = *plMultiplicand;
    lResult = lOriginal * lMultiplier;
  } while (InterlockedCompareExchange(plMultiplicand,
                                      lResult, lOriginal) != lOriginal);
  return lResult;
}
```

[Typo in algorithm fixed 9:00am.]

To perform a complicated function on the multiplicand, we perform three steps.

First, capture the value from memory: `lOriginal = *plMultiplicand;`

Second, compute the desired result from the captured value: `lResult = lOriginal * lMultiplier;`

Third, store the result provided the value in memory has not changed: `InterlockedCompareExchange(plMultiplicand, lResult, lOriginal)`

If the value did change, then this means that the interlocked operation was unsucessful because somebody else changed the value while we were busy doing our computation. In that case, loop back and try again.

If you walk through the scenario above with this new InterlockedMultiply function, you will see that after the interloping InterlockedIncrement, the loop will detect that the value of “x” has changed and restart. Since the final update of “x” is performed by an InterlockedCompareExchange operation, the result of the computation is trusted only if “x” did not change value.

Note that this technique works only if the operation being performed is a pure function of the memory value and the function parameters. If you have to access other memory as part of the computation, then this technique will not work! That’s because those other memory locations might have changed during the computation and you would have no way of knowing, since InterlockedCompareExchange checks only the memory value being updated.

Failure to heed the above note results in problems such as the so-called “ABA Problem”. I’ll leave you to google on that term and read about it. Fortunately, everybody who talks about it also talks about how to **solve** the ABA Problem, so I’ll leave you to read that, too.

Once you’ve read about the ABA Problem and its solution, you should be aware that the solution has already been implemented for you, via the Interlocked SList functions.

Raymond Chen

Follow