An extensible interlocked arithmetic function (via lambdas)

devblogs.microsoft.com/oldnewthing/20140516-00

May 16, 2014

Raymond Chen

Some time ago, I noted that you can build other interlocked operations out of `InterlockedCompareExchange`. Here's an example:

```
using System.Threading;
public static int InterlockedMax(ref int location, int value)
{
    int initialValue, newValue;
    do
    {
        initialValue = location;
        newValue = Math.Max(initialValue, value);
    }
    while (Interlocked.CompareExchange(ref location, newValue,
                                       initialValue) != initialValue);
    return initialValue;
}
```

(There's a corresponding C++ version, which I leave as an exercise.)

This function atomically updates a "highest value seen so far" variable. It follows the usual pattern:

- Capture the starting value.
- Do a computation based on that value.
- Compare-exchange the new value in.
- If the compare-exchange failed, then start over.

(For bonus points, add an early-out if the operation should be abandoned.)

You can make this function extensible by use of lambdas, so that you can update the old value with any computation you like.

```
using System;
using System.Threading;
public static int InterlockedCombine(ref int location,
                                     Func<int, int> update)
{
    int initialValue, newValue;
    do
    {
        initialValue = location;
        newValue = update(initialValue);
    }
    while (Interlocked.CompareExchange(ref location, newValue,
                                       initialValue) != initialValue);
    return initialValue;
}
public static int InterlockedMax(ref int location, int value)
{
    return InterlockedCombine(ref location,
                              v => Math.Max(v, value));
}
public static int InterlockedMultiply(ref int location, int value)
{
    return InterlockedCombine(ref location,
                              v => v * value);
}
public static int InterlockedIncrementWithSaturation(
    ref int location, int maximum)
{
    return InterlockedCombine(ref location,
                              v => v < maximum ? v + 1 : maximum);
}
public static int InterlockedCompareExchangeIfNotEqual(
    ref int location, int newValue, int avoidValue)
{
    return InterlockedCombine(ref location,
                              v => v != avoidValue ? newValue : avoidValue);
}
```

Raymond Chen

Follow