The performance improvements of a lock-free algorithm is often not in the locking

devblogs.microsoft.com/oldnewthing/20110421-00

April 21, 2011

Raymond Chen

GWO wonders what the conditions are under which the lock-free version significantly outpeforms a simple critical section.

Remember that switching to a lock-free algorithm should be guided by performance measurements. Switching from a simple algorithm to a complex one shouldn't be done unless you know that the simple algorithm is having trouble.

That said, here are some non-obvious advantages of a lock-free algorithm over one that uses a simple lock. (Later, we'll see how you can take advantage of these techniques without actually going lock-free.)

Consider a program that uses a simple critical section to perform something like the singleton constructor. Instead of a fancy lock-free algorithm, we use the much simpler version:

```
CRITICAL_SECTION g_csSingletonX;
X *g_px = NULL;
X *GetSingletonX()
{
    EnterCriticalSection(&g_csSingletonX);
    if (g_px == NULL)
    {
        g_px = new(nothrow) X();
    }
    LeaveCriticalSection(&g_csSingletonX);
    return g_px;
}
```

This simple code can run into trouble if the constructor function itself requires some locks, because now you have to impose a lock hierarchy in order to avoid a deadlock. (And this becomes impossible if the constructor function belongs to code outside your control.)

When working out what your lock hierarchy should be, you may discover that you need to consolidate some locks. This avoids the inversion problem, but it also reduces your lock granularity. You might decide to use a single lock to cover all singletons, and then you later discover that you also have to extend the lock that protects X's constructor to cover other operations on X.

```
CRITICAL_SECTION g_csCommon;
// (updated to remove double-check lock because that just raises
// more questions that distract from the point of the article)
X *GetSingletonX()
{
    EnterCriticalSection(&g_csCommon);
    if (g_px == NULL)
    {
        g_px = new(nothrow) X();
    }
    LeaveCriticalSection(&g_csCommon);
    return g_px;
}
Y *GetSingletonY()
{
    EnterCriticalSection(&g_csCommon);
    if (g_py == NULL)
    {
        g_py = new(nothrow) Y();
    }
    LeaveCriticalSection(&g_csCommon);
    return g_py;
}
void X::DoSomething()
{
    EnterCriticalSection(&g_csCommon);
    .. something ..
    LeaveCriticalSection(&g_csCommon);
}
```

Over time, your quiet little singleton lock has turned into a high-contention lock in your system.

One nice thing about a lock-free algorithm is that since there is no lock, it can't create inversion in a lock hierarchy. (Of course, you have to be careful not to use the interlocked operations to build a private lock, because that puts you back where you started.)

Another nice consequence of a lock-free algorithm is that, since there is no lock, you don't have to handle the `WAIT_ABANDONED` case. The data structure is never inconsistent; it passes atomically from one consistent state to another. Therefore, there's no need to write code to clean up leftover inconsistency. This came in handy in a case we looked at earlier, so that an application which crashes at an inopportune time will not corrupt the shared data and require a server reboot.

Raymond Chen

Follow