Slim reader/writer locks don't remember who the owners are, so you'll have to find them some other way

devblogs.microsoft.com/oldnewthing/20110810-00

August 10, 2011

Raymond Chen

The slim reader/writer lock is a very convenient synchronization facility, but one of the downsides is that it doesn't keep track of who the current owners are. When your thread is stuck waiting to acquire a slim reader/writer lock, a natural thing to want to know is which threads own the resource your stuck thread waiting for.

Since there's not facility for going from the waiting thread to the owning threads, you'll just have to find the owning threads some other way. Here's the thread that is waiting for the lock in shared mode:

```
ntdll!ZwWaitForKeyedEvent+0xc
ntdll!RtlAcquireSRWLockShared+0x126
dbquery!CSearchSpace::Validate+0x10b
dbquery!CSearchSpace::DecomposeSearchSpace+0x3c
dbquery!CQuery::AddConfigs+0xdc
dbquery!CQuery::ResolveProviders+0x89
dbquery!CResults::CreateProviders+0x85
dbquery!CResults::GetProviders+0x61
dbquery!CResults::CreateResults+0x11c
```

Okay, how do you find the thread that owns the lock?

First, slim reader/writer locks are usable only within a process, so the candidate threads are the one within the process.

Second, the usage pattern for locks is nearly always something like

```
enter lock
do something
exit lock
```

It is highly unusual for a function to take a lock and exit to external code with the lock held. (It might exit to other code within the same component, transferring the obligation to exit the lock to that other code.) Therefore, you want to look for threads that are still inside

`dbquery.dll`, possibly even still inside `CSearchSpace` (if the lock is a per-object lock rather than a global one).

Of course, the possibility might be that the code that entered the lock messed up and forgot to release it, but if that's the case, no amount of searching for it will find anything since the culprit is long gone. Since debugging is an exercise in optimism, we may as well proceed on the assumption that we're not in the case. If it fails to find the lock owner, then we may have to revisit the assumption.

Finally, the last trick is knowing which threads to ignore. For now, you can also ignore the threads that are waiting for the lock, since they are the victims not the cause. (Again, if we fail to find the lock owner, we can revisit the assumption that they are not the cause; for example, they may be attempting to acquire the lock recursively.)

As it happens, there is only one thread in the process that passes all the above filters.

```
dbquery!CProp::Marshall+0x3b
dbquery!CRequest::CRequest+0x24c
dbquery!CQuery::Execute+0x668
dbquery!CResults::FillParams+0x1c4
dbquery!CResults::AddProvider+0x4e
dbquery!CResults::AddConfigs+0x1c5
dbquery!CResults::CreateResults+0x145
```

This may not be the source of the problem, but it's a good start. (Actually, it looks very promising since the problem is probably that the process on the other side of the marshaller is stuck.)

Raymond Chen

Follow