If the slim reader/writer lock (SRWLOCK) doesn't remember who the shared lock owner is, does that mean it's okay to acquire it recursively?

devblogs.microsoft.com/oldnewthing/20220304-00

March 4, 2022

Raymond Chen

The slim reader/writer lock (SRWLOCK) synchronization object was introduced in Windows Vista and allows threads to acquire the lock either in shared or exclusive modes. Multiple simultaneous shared acquisitions are permitted from different threads, but an exclusive acquisition cannot coexist with any other acquisition (either shared or exclusive).

Some sacrifices had to be made in order for the slim reader/writer lock to be small and fast. It doesn't support lock upgrades or downgrades (converting a shared lock to exclusive or vice versa), or recursive acquisition (acquiring a lock when the calling thread already possesses a lock).

Since the slim reader/writer lock is the size of a pointer, and a successful shared acquisition returns control to the caller, this means that all of the state for recording shared lock acquisitions must fit into a single pointer-sized variable. In particular, there is not enough room to record all of the threads that have acquired the lock in shared mode. I've seen it argued that this proves that it's actually okay to acquire the lock recursively in shared mode, since there's no way the operating system could detect that you broke the rules. Besides, shared locks can be taken by multiple threads concurrently, so why should concurrent acquisition by the same thread be any different?

While it's true that there's no way the operating system can detect that you broke the rules, there is no requirement that the operating system do this detection. There can be other consequences of breaking the rules.

Although slim reader/writer locks are neither fair nor FIFO, the current implementation is *mostly* fair and FIFO. If the lock is held in shared mode and there is an exclusive waiter, then shared locks queue up behind the exclusive waiter rather than piggybacking off the existing shared wait. (We followed the same policy when we wrote our asynchronous version of a slim reader/writer lock.) This avoids a situation where a highly-contended resource is constantly being acquired briefly by multiple threads in shared mode, with overlapping acquisition lifetimes, causing the exclusive acquirer to be locked out indefinitely.

Thread 1	Thread 2	Thread 3	Thread 4
	Acquired shared		
Try acquire exclusive		Acquired shared	
(still waiting)			Acquired shared
	Acquired shared		
		Acquired shared	
(still waiting)			Acquired shared
	Acquired shared		
		Acquired shared	

To avoid this problem, the SRWLock makes a shared acquisition attempt queue up behind any pending exclusive waiter. In the above scenario, we get this:

Thread 1	Thread 2	Thread 3	Thread 4
	Acquired shared		
Try acquire exclusive		Try acquire shared (still waiting)	
Acquired exclusive			Try acquire shared (still waiting)
	Acquired shared	Acquired shared	Acquired shared

Now you can see what can go wrong if you attempt a recursive shared acquisition: Your second shared acquisition waits for the pending exclusive waiter. The pending exclusive waiter is waiting for the first shared acquisition to be released. And the first shared acquisition can't be released until it regains control of the thread from the second shared acquisition. Three-party circular deadlock.

Thread 1		Thread 2
		Acquired shared
Try acquire exclusive	⇗	
(still waiting)	⇐	Try acquire shared

The Application Verifier detects and reports attempted recursive acquisitions to help you find them before they cause problems in production. As I noted some time ago, one of my colleagues remarked,

> We hit a deadlock in production due to erroneous recursive acquisition. It is fiendishly difficult to debug. I would put it in the "immediate fix" category.

Bonus chatter: Recursive exclusive acquisition results in a consistent deadlock, so you might wonder why Application Verifier goes to the effort of diagnosing it, seeing as you'd certainly notice that your program is hung.

First of all, Application Verifier records additional information to help you diagnose the problem: It captures the stack at the point the first acquisition was made. This additional information may help you identify the root cause. For example, maybe there's a code path out of a function that forgets to release the lock. The acquisition stack trace may help you find that function.

Second, you may be deadlocking without realizing it. For example, if the deadlock occurs during thread cleanup, you may not notice that your thread never exits. It just manifests itself as a thread leak. Furthermore, if there was any other cleanup code that is expected to be running on the thread after the erroneous deadlock, then that cleanup code will never run. These leaks may go unnoticed until you find that your production system's memory usage slowly increases and its performance slowly decreases, until it finally hits a resource exhaustion failure after being left running for days. Bugs that require days to reproduce are not anybody's idea of fun.

Raymond Chen

Follow