The case of the SRWLock violation in a thread pool work item

devblogs.microsoft.com/oldnewthing/20180202-00

February 2, 2018

Raymond Chen

A customer encountered a failure detected by Application Verifier and wanted some assistance understanding what happened and how they can fix it.

The verifier failure was “invalid SRWLock owner”, and the failure details were as follows:

- Lock = (the address of an SRWLock)
- Local thread ID = (the thread trying to release the lock)
- Owner thread ID = (some thread pool thread)
- Acquire stack trace = (see below)

The acquire stack trace looked like this:

```
verifier!AVrfpRtlAcquireSRWLockShared+0x5e
contoso!Microsoft::WRL::Wrappers::SRWLock::LockShared+0x16
contoso!CDoodad::NotifyListenersAsync+0xb2
contoso!CDoodad::OnPropertyChanged+0x20a
contoso!CChannel::ProcessNotification+0x20a
ntdll!RtlpTpWaitCallback+0x9b
ntdll!TppExecuteWaitCallback+0x9b
ntdll!TppDirectExecuteCallback+0xb9
ntdll!TppWorkerThread+0x497
kernel32!BaseThreadInitThunk+0x14
ntdll!RtlUserThreadStart+0x21
```

The thread pool processed a notification from the channel, and the channel handed the property-change notification to the doodad, and the doodad took a shared lock so it could notify the listeners.

Here’s the stack that is trying to release the lock:

```
ntdll!RtlReportException+0x9d
verifier!VerifierCaptureContextAndReportStop+0xc0
verifier!VerifierStopMessageEx+0x750
verifier!AVrfpVerifySRWLockRelease+0x126
verifier!AVrfpRtlReleaseSRWLockShared+0x42
contoso!Microsoft::WRL::Wrappers::Details::SyncLockShared::{dtor}+0x12
contoso!Doodad::Notifier::`scalar deleting destructor'+0x20
contoso!Doodad::Notifier::DispatchNotification+0x11a0d
ntdll!TppWorkpExecuteCallback+0x131
ntdll!TppWorkerThread+0x43e
kernel32!BaseThreadInitThunk+0x14
ntdll!RtlUserThreadStart+0x21
```

The customer was kind enough to share the code for the classes in question. Here's what the code is trying to do: When the doodad learns from the channel that a property has changed, it notifies all the listeners who subscribed to that change. To avoid blocking the channel, the code queues a work item to deliver the notifications.

But there's a race condition: What if the doodad is destroyed while the work item is still waiting in the thread pool? When the work item gets to run on the thread pool, it will try to access a freed doodad. (You also have this problem if the doodad is destroyed while the work item is running.)

The code solves this problem like this:

```
// Code in italics is wrong.
class Doodad
{
 ...
private:
  ~Doodad()
  {
    auto ensureNoNotifications = m_notifyLock.LockExclusive();
    ... other cleanup ...
  }

  class Notifier
  {
  public:
    static void QueueNotification(CDoodad* doodad)
    {
      auto notifier = std::make_unique<Notifier>(doodad);
      if (!QueueUserWorkItem(DispatchNotification, notifier.get(), 0)) {
        throw some_error();
      }
      notifier.release(); // work item owns the Notifier now
    }

  private:
    Notifier(CDoodad* doodad) :
      m_doodad(doodad),
      m_sharedLock(doodad->m_notifyLock.LockShared())
    {
    }

    static DWORD CALLBACK DispatchNotification(void* parameter)
    {
      // ensure that the Notifier gets deleted
      auto notifier = std::make_unique<Notifier>(
                        reinterpret_cast<Notifier*>(parameter));

      ... notify the listeners ...
    }

  private:
    CDoodad* m_doodad;
    Microsoft::WRL::Wrappers::SRWLock::SyncLockShared m_sharedLock;
  };

  void NotifyListenersAsync()
  {
    Notifier::QueueNotification();
  }

  Microsoft::WRL::Wrappers::SRWLock m_notifyLock;
};
```

In words: When we want to notify the listeners, we create a `Notifier` object and queue it onto the thread pool. The `Notifier` object has a reference to the `Doodad`, as well as a shared lock. Since a shared lock can be acquired multiple times, this means that multiple notifications can be in flight.

When the work item is dispatched, it notifies all the listeners and then destroys the `Notifier`, which in turn releases the shared lock.

Finally, when the `Doodad` is destructed, it takes an exclusive lock. Since an exclusive lock cannot be acquired while there are still any shared locks active, this will wait until all of the notification work items have been retired. That way, no notification will operate on a destroyed `Doodad`.

Do you see the problem?

If you know what's good for you, you put RAII object that represents a lock on the stack, or inside an object whose lifetime is tied to the stack, because the lifetime of the lock needs to be tied to the thread.

But this code doesn't do what proper deity-fearing code does. Instead, it saves the lock in a member variable of an object that is not destroyed before the function returns. This means that the lifetime of the shared lock is not tied to a thread. The shared lock is acquired by the thread that queues the work item, and the shared lock is released by the thread that processes the work item. If you're lucky, they are the same thread and nobody gets hurt. But if you're not lucky, then they are different threads, and you violated the lock rules and have entered the world of undefined behavior.

The developer here thought they were being clever by abusing the `SRWLock`, but in fact they were getting themselves into trouble.

To fix the problem, they switched to a custom synchronization object built out of `WaitOnAddress`.

```
class Doodad
{
 ...
private:
  ~Doodad()
  {
    WaitForValueByAddress(m_notificationCount,
      [](auto&& value) { return value == 0; });
    ... other cleanup ...
  }

  class Notifier
  {
  public:
    static void QueueNotification(CDoodad* doodad)
    {
      auto notifier = std::make_unique<Notifier>(doodad);
      if (!QueueUserWorkItem(DispatchNotification, notifier.get(), 0)) {
        throw some_error();
      }
      notifier.release(); // work item owns the Notifier now
    }

  private:
    Notifier(CDoodad* doodad) : m_doodad(doodad)
    {
      InterlockedIncrement(&doodad->m_notificationCount);
    }

    ~Notifier()
    {
      if (InterlockedDecrement(&doodad->m_notificationCount) == 0) {
        WakeByAddressSingle(&doodad->m_notificationCount);
      }
    }

    // Not copyable or movable
    Notifier(const Notifier&) = delete;
    Notifier(Notifier&&) = delete;
    Notifier& operator=(const Notifier&) = delete;
    Notifier& operator=(Notifier&&) = delete;

    static DWORD CALLBACK DispatchNotification(void* parameter)
    {
      // ensure that the Notifier gets deleted
      auto notifier = std::make_unique<Notifier>(
                        reinterpret_cast<Notifier*>(parameter));

      ... notify the listeners ...
    }

  private:
```

```
    CDoodad* m_doodad;
  };

  void NotifyListenersAsync()
  {
    Notifier::QueueNotification();
  }

  LONG m_notificationCount = 0;
};
```

`WaitOnAddress` requires Windows 8 or higher, and the customer was okay with that. If the customer needed something that ran on Windows Vista, they could have accomplished something similar with a condition variable.

```
class Doodad
{
 ...
private:
  ~Doodad()
  {
    auto lock = m_notifierLock.LockExclusive();
    SleepConditionVariableExclusiveSRWUntil(&m_notifierCV,
        &m_notifierLock,
        [](auto&& value) { return value == 0; });
    ... other cleanup ...
  }

  class Notifier
  {
  public:
    static void QueueNotification(CDoodad* doodad)
    {
      auto notifier = std::make_unique<Notifier>(doodad);
      if (QueueUserWorkItem(DispatchNotification, this, 0)) {
        notifier.release(); // work item owns the Notifier now
      }
      throw some_error();
    }

  private:
    Notifier(CDoodad* doodad) : m_doodad(doodad)
    {
      auto lock = m_doodad->m_notifierLock.LockExclusive();
      &doodad->m_notificationCount++;
    }

    ~Notifier()
    {
      auto lock = m_doodad->m_notifierLock.LockExclusive();
      if (--doodad->m_notificationCount == 0) {
        WakeConditionVariable(&doodad->m_notificationCV);
      }
    }

    // Not copyable or movable
    Notifier(const Notifier&) = delete;
    Notifier(Notifier&&) = delete;
    Notifier& operator=(const Notifier&) = delete;
    Notifier& operator=(Notifier&&) = delete;

    static DWORD CALLBACK DispatchNotification(void* parameter)
    {
      // ensure that the Notifier gets deleted
      auto notifier = std::make_unique<Notifier>(
                        reinterpret_cast<Notifier*>(parameter));
```

```
      ... notify the listeners ...
    }

  private:
    CDoodad* m_doodad;
  };

  void NotifyListenersAsync()
  {
    Notifier::QueueNotification();
  }

  LONG m_notificationCount = 0;
  Microsoft::WRL::Wrappers::SRWLock m_notificationLock;
  CONDITION_VARIABLE m_notificationCV = CONDITION_VARIABLE_INIT;
};
```

Raymond Chen

Follow