The case of the recursively-acquired non-recursive lock, and how to avoid the unintentional reentrancy

devblogs.microsoft.com/oldnewthing/20220902-00

September 2, 2022

Raymond Chen

A customer encountered a deadlock due to unexpected reentrancy, and they were looking for guidance in fixing it.

Here's the code in question:

```
struct WidgetTracker : IWidgetChangeNotificationSink
{
    /* other stuff not relevant here */

    /// IWidgetChangeNotificationSink
    STDMETHODIMP OnCurrentWidgetChanged();

private:
    WRL::ComPtr<IWidget> m_currentWidget;
    std::mutex m_mutex;
};

HRESULT WidgetTracker::OnCurrentWidgetChanged()
{
    auto guard = std::lock_guard(m_mutex);
    RETURN_IF_FAILED(GetCurrentWidget(&m_currentWidget));
    return S_OK;
}
```

The idea here is that the `WidgetTracker` listens for notifications that the current widget has changed, and when it receives that notification, it updates its local cache to hold the new current widget.

The hang occurred with this stack:

```
ntdll!ZwWaitForAlertByThreadId
ntdll!RtlAcquireSRWLockExclusive
contoso!WidgetTracker::OnCurrentWidgetChanged
rpcrt4!Invoke
rpcrt4!Ndr64StubWorker
rpcrt4!NdrStubCall3
combase!CStdStubBuffer_Invoke
combase!InvokeStubWithExceptionPolicyAndTracing::__l6::<lambda_...>::operator()
combase!ObjectMethodExceptionHandlingAction<<lambda_...> >
combase!InvokeStubWithExceptionPolicyAndTracing
combase!DefaultStubInvoke
combase!SyncServerCall::StubInvoke
combase!StubInvoke
combase!ServerCall::ContextInvoke
combase!DefaultInvokeInApartment
combase!ReentrantSTAInvokeInApartment
combase!ComInvokeWithLockAndIPID
combase!ThreadDispatch
combase!ThreadWndProc
user32!UserCallWinProcCheckWow
user32!DispatchMessageWorker
combase!CCliModalLoop::MyDispatchMessage
combase!CCliModalLoop::PeekRPCAndDDEMessage
combase!CCliModalLoop::BlockFn
combase!ModalLoop
combase!ThreadSendReceive
combase!CSyncClientCall::SwitchAptAndDispatchCall
combase!CSyncClientCall::SendReceive2
combase!SyncClientCallRetryContext::SendReceiveWithRetry
combase!CSyncClientCall::SendReceiveInRetryContext
combase!ClassicSTAThreadSendReceive
combase!CSyncClientCall::SendReceive
combase!CClientChannel::SendReceive
combase!NdrExtpProxySendReceive
rpcrt4!Ndr64pSendReceive
rpcrt4!NdrpClientCall3
combase!ObjectStublessClient
combase!ObjectStubless
litware!Widget::~Widget
litware!Widget::`scalar deleting destructor'
litware!Widget::Release
contoso!Microsoft::WRL::ComPtr<IWidget>::InternalRelease
contoso!Microsoft::WRL::ComPtr<IWidget>::ReleaseAndGetAddressOf
contoso!Microsoft::WRL::Details::ComPtrRef<...>::operator struct IWidget **
contoso!WidgetTracker::OnCurrentWidgetChanged
rpcrt4!Invoke
rpcrt4!Ndr64StubWorker
rpcrt4!NdrStubCall3
combase!CStdStubBuffer_Invoke
combase!InvokeStubWithExceptionPolicyAndTracing::__l6::<lambda_...>::operator()
combase!ObjectMethodExceptionHandlingAction<<lambda_...> >
```

```
combase!InvokeStubWithExceptionPolicyAndTracing
...
```

Reading from the bottom up, what happened is that the current widget changed, and the `WidgetTracker` received the change notification. The `WidgetTracker` locks the mutex, and then wants to get the new current `Widget`, but first it releases the old `Widget`.

It is that release of the old widget that causes trouble, because it makes a cross-process call, and while waiting for the cross-process call to complete, the current widget changes again, and the `OnCurrentWidgetChanged` method gets called again. (It is evident that this code is running on a single-threaded apartment. If it were running in a multi-threaded apartment, the second call would have arrived on a different thread.)

The problem is that we are releasing our reference to the old widget while holding a lock, and that creates the opportunity for mayhem, since we don't control what the widget will do when it is released. And if this is the final release of the widget, it will probably do a lot of work.

This is another case of the hidden callout: The destructor.

And the solution is the same: Destruct the reference to the old widget outside the lock.

```
HRESULT WidgetTracker::OnCurrentWidgetChanged()
{
    WRL::ComPtr<IWidget> widget;
    auto guard = std::lock_guard(m_mutex);
    RETURN_IF_FAILED(GetCurrentWidget(&widget));
    m_currentWidget.Swap(widget);
    return S_OK;
}
```

We declare a `ComPtr<IWidget>` before taking the lock, so that it destructs after the lock is released. (Remember that in C++, local variables are destructed in reverse order of construction.) After we get the current widget into the local `widget`, we swap it with the old one, and then return.

The lock guard destructs first, which exits the lock. and then the `ComPtr<IWidget>` destructs, which releases the old widget. This release occurs outside the lock, so any re-entrancy is not going to create a deadlock.

Raymond Chen

Follow