One way to defer work when a re-entrant call is detected

devblogs.microsoft.com/oldnewthing/20230426-00

April 26, 2023

Raymond Chen

A customer had a single-threaded COM object that was experiencing unexpected re-entrancy.

```
enum WidgetState;

struct MyObject : public IWidgetStateListener
{
    /* ... */

    // IWidgetStateListener
    HRESULT OnStateChanged(WidgetState newState);

    /* ... */
    WidgetState m_widgetState = WidgetState::Normal;
    WidgetStateMonitor m_monitor;
};

HRESULT MyObject::OnStateChanged(WidgetState newState)
{
    m_monitor.Disconnect();

    m_widgetState = newState;

    m_monitor.Connect(newState);

    return S_OK;
}
```

The catch was that disconnecting the monitor involved a cross-thread COM call.

When a single-threaded COM apartment makes a call to another thread, it allows inbound calls to be made while waiting for the call to complete. This is necessary so that if the recipient of the outbound call tries to call back, the call can come back in. For example, the thread might call `IPersistStream::Load` on an external object. As part of the `IPersistStream::Load`, the external object will naturally read from the stream. If the call to `IStream::Read` were not allowed through, then the system would deadlock.

However, in this case, the inbound call was not coming from the `m_monitor` object. Rather, while waiting for that external object to finish disconnecting, the widget state changed *again*. This caused `OnStateChanged` to be called while a previous call was still outstanding, resulting in re-entrancy and mass confusion.

`OnStateChanged(state1)`
`m_monitor.Disconnect();`
re-entrant call while waiting for `Disconnect`

`OnStateChanged(state2);`
`m_monitor.Disconnect();` ← Oops, double-disconnect
`m_widgetState = state2;`
`m_monitor.Connect(state2);`
`OnStateChanged(state2)` returns

`m_widgetState = state1;` `m_monitor.Connect(state1);` ← Oops, forgot to disconnect state2 `OnStateChanged(state1)` returns

Okay, so how can we fix this?

It so happens that the nature of the state changes is that the `MyObject` cares only about the final state of the widget. This means that multiple state change notifications can be coalesced, which simplifies our solution.

The last bit that is important is that the `MyObject` does not have to complete all its processing immediately upon receiving a state change notification. As long as the final state is eventually processed, the object will work okay.

When re-entrancy is detected, we don't want to start processing the new change notification immediately, because the old one is still active. Instead, we'll just record the value for later processing when things are safe.

```
struct MyObject : public IWidgetStateListener
{
    /* ... */

    // IWidgetStateListener
    HRESULT OnStateChanged(WidgetState newState);

    /* ... */
    WidgetState m_widgetState = WidgetState::Normal;

    // New members
    bool m_updatingState = false;
    bool m_updatePostponed = false;
    WidgetState m_postponedState;
};

HRESULT MyObject::OnStateChanged(WidgetState newState)
{
    if (m_updatingState) {
        m_updatePostponed = true;
        m_postponedState = newState;
        return S_OK;
    }

    m_updatingState = true;

    m_monitor.Disconnect();

    m_widgetState = newState;

    m_monitor.Connect(newState);

    m_updatingState = false;
    // Not finished yet

    return S_OK;
}
```

The other half of the solution is arranging that the already-running copy of `OnState-Changed`, after it finishes it work, checks whether an update occurred while it was working. If so, then go back and process that postponed update.

```
HRESULT MyObject::OnStateChanged(WidgetState newState)
{
    if (m_updatingState) {
        m_updatePostponed = true;
        m_postponedState = newState;
        return S_OK;
    }

    m_updatingState = true;

    do {
        m_updatePostponed = false;

        m_monitor.Disconnect();

        m_widgetState = newState;

        m_monitor.Connect(newState);

        newState = m_postponedState;
    } while (m_updatePostponed);
    m_updatingState = false;

    return S_OK;
}
```

After setting the `m_updatingState` flag to `true`, we clear the `m_updatePostponed` state, and then start doing our work. If re-entrancy occurs during our work, the re-entrant call will see that `m_updatingState` is `true`, and instead of doing the work immediately, it sets the `m_updatePostponed` flag to `true` and remembers the postponed value.

After the work completes, we check whether there was a postponed update. If so, we loop back and process that postponed update.

If your policy is that all changes must be processed, rather than dropping intermediate changes, you'll have to keep a queue of pending changes rather than just remembering the last one. I'll leave that as an exercise.

Note also that I'm assuming that this is a single-threaded object, so I don't have to worry about concurrency or unfairness. Concurrency is not a problem since there is no multi-threaded access to `MyObject`. Unfairness is not a problem because the costs are all being paid by the same thread; all we did was shift the time the cost is paid.