Processing a ValueSet or PropertySet even in the face of possible mutation, part 2

devblogs.microsoft.com/oldnewthing/20220713-00

July 13, 2022

Raymond Chen

Last time, we looked at one way of saving the contents of a ValueSet or PropertySet while remaining resilient to concurrency modification: After each mutation, you try to save the updated collection, but bail out if you notice a subsequent modification.

Another model for this concurrency problem would be to have the `Save` function just keep looping until it is able to save successfully. It would also check if another thread was already saving, in which case the second thread could just return immediately, knowing that the first thread will eventually get everything saved.

```
void MyPropertySet::Save()
{
    while (guard = std::unique_lock(m_lock, std::try_to_lock)) {
        try {
            SomeKindOfDataBuffer buffer;
            auto it = m_propertySet.First();
            if (it.HasCurrent()) {
                do {
                    buffer.AddKeyAndValue(it.Key(), it.Value());
                } while (it.MoveNext());
            }

            SaveToFile(buffer);
            it.HasCurrent(); // verify that the collection is still unchanged
            return;
        } catch (winrt::hresult_changed_state const&) {
            // Didn't work.
        }
    }
}
```

Here, the lock is a “busy” flag. The first thread to save becomes the “saving” thread, and it keeps trying to save the data until it makes it all the way to the end without the property set being mutated. Note that we make one final check after saving the results to the file, to catch the case where the property set is mutated while we are saving to disk.

In this model, the thread that first modifies the property set will keep trying to save until it succeeds. If the property set is constantly being mutated by other threads, then those other thread keep invalidating the work of the first thread, which ends up looping back to try again.

I don't like this version because it means that the time it takes to insert a value into the property set is potentially unbounded if you have competing threads repeatedly mutating the property set and preventing the `Save` function from succeeding.

On the other hand, this minor change would work:

```
winrt::fire_and_forget MyPropertySet::Save()
{
    auto lifetime = get_strong();
    co_await resume_background();

    while (guard = std::unique_lock(m_lock, std::try_to_lock)) {
        try {
            SomeKindOfDataBuffer buffer;
            auto it = m_propertySet.First();
            if (it.HasCurrent()) {
                do {
                    buffer.AddKeyAndValue(it.Key(), it.Value());
                } while (it.MoveNext());
            }

            SaveToFile(buffer);
            it.HasCurrent(); // verify that the collection is still unchanged
            return;
        } catch (winrt::hresult_changed_state const&) {
            // Didn't work.
            // Optionally back off before trying again.
        }
    }
}
```

This time, the work is kick off to a background thread so that the thread which mutated the property set is released back to the caller immediately. With this change, the pattern works out better because there is now only one thread trying to save the property set, and it just keeps retrying until it succeeds, rather than having multiple threads all trying to save, with only one of them succeeding and the rest having done useless work. In this model, if there are a thousand changes in a row, we aren't likely to try to serialize a thousand times. Each time the property set mutates, we abandon the current operation and start over. We can add a back-off period so we don't get forced to restart quite as often.

Next time, we'll look at some other implementation details of this pattern.

Raymond Chen

Follow