Adding state to the update notification pattern, part 4

devblogs.microsoft.com/oldnewthing/20240422-00

April 22, 2024

Raymond Chen

Last time, we developed a stateful but coalescing update notification, and we noted that the UI thread was doing a lot of heavy lifting. What if you don't have a UI thread to do implicit serialization for you?

If there were no `resume_foreground(Dispatcher())`, we would have a race if a `TextChanged` occurs after the worker has decided to exit, but before it has had a chance to mark itself as not busy. Here's an alternate version that demonstrates the race.

```
class EditControl
{
    ⟦ ... existing class members ... ⟧

    std::atomic<bool> m_busy;
    std::mutex m_mutex;
    std::optional<string> m_pendingText;
};

winrt::fire_and_forget
EditControl::TextChanged(std::string text)
{
    auto lifetime = get_strong();

    ExchangePendingText(std::move(text));
    if (m_busy.exchange(true)) {
        co_return;
    }

    co_await winrt::resume_background();

    while (auto pendingText = ExchangePendingText(std::nullopt);
           pendingText) {
        auto matches = BuildMatches(*pendingText);

        if (matches) {
            SetAutocomplete(*matches);
        }
    }
    m_busy = false;
}
```

In this alternate version, the `m_mutex` is critical because the background thread picks up the `m_pendingText` for the next iteration. But now there is a race window if a change to the pending text occurs immediately after we notice that there is no pending text and before we clear the busy flag.

UI thread	Background thread

`TextChanged("Bob")`
`ExchangePendingText("Bob")`
`m_busy = true;`
`resume_background()`	

	`BuildMatches("Bob");`
	`SetAutocomplete(*matches)`
	No pending text, so exit `while` loop

`TextChanged("Alice");`
`ExchangePendingText("Alice")`
`m_busy` already `true`
`co_return;`	

	`m_busy = false;`

To avoid this race, `m_busy` needs to move under the mutex. And once it’s moved under the mutex, it doesn’t need to be atomic any more. The need to extend the scope of the mutex means that our cute little helper functions won’t really cut it any more. We’ll have to manage the locks ourselves.

```
class EditControl
{
    ⟦ ... existing class members ... ⟧

    bool m_busy = false;
    std::mutex m_mutex;
    std::optional<string> m_pendingText;
};
winrt::fire_and_forget
EditControl::TextChanged(std::string text)
{
    auto lifetime = get_strong();

    {
        auto lock = std::unique_lock(m_mutex);
        m_pendingText = std::move(text);
        if (std::exchange(m_busy, true)) {
            co_return;
        }
    }

    co_await winrt::resume_background();

    while (true) {
        {
            auto lock = std::unique_lock(m_mutex);
            if (!m_pendingText) {
                m_busy = false;
                co_return;
            }
            text = std::move(*m_pendingText);
            m_pendingText.release();
        }

        auto matches = BuildMatches(text);

        if (matches) {
            SetAutocomplete(*matches);
        }
    }
}
```

Next time, we’ll solve the same problem using a different approach.