Adding state to the update notification pattern, part 2

devblogs.microsoft.com/oldnewthing/20240418-00

April 18, 2024

Raymond Chen

Last time, we started looking at solving the problem of a stateful but coalescing update notification, where multiple requests for work can arrive, and your only requirement is that you send a notification for the last one. Any time a new request for work arrives, it replaces the existing one.

One attempt to fix this is to check if the work is already in progress, and if so, then hand off the new query to the existing worker. We are using `winrt::fire_and_forget`, which fails fast on any unhandled exception. This saves us from having to worry about recovering from exceptions. (At least for now.)

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

    std::mutex m_workMutex;
    std::mutex m_textMutex;
    std::optional<string> m_pendingText;
};

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

    auto workLock = std::unique_lock(m_workMutex, std::try_to_lock);
    if (!workLock) {
        auto textLock = std::unique_lock(m_textMutex);
        m_pendingText = std::move(text);
        co_return;
    }

    while (true) {
        co_await winrt::resume_background();

        std::vector<std::string> matches;
        for (auto&& candidate : FindCandidates(text)) {
            if (candidate.Verify()) {
                matches.push_back(candidate.Text());
            }
        }

        co_await winrt::resume_foreground(Dispatcher());

        SetAutocomplete(matches);

        auto text = std::unique_lock(m_textMutex);
        if (!m_pendingText) {
            co_return;
        }
        text = std::move(*m_pendingText);
        m_pendingText.reset();
    }
}
```

But before even thinking about whether this addresses the race condition, we have to call out that this code isn't even legal.

This code carries a lock across a suspension point, which we saw is not a good idea. In this case, we use `try_` mode to acquire the mutex, and the rules for `try_lock` say two things, one bad, and the other worse.

The bad thing is that `try_lock` is allowed to fail spuriously and return `false` even if the mutex is not locked. This means that it's possible that the `workLock` will report that it does not own the lock, even though the lock is available, and you will have a `m_pendingText` sitting around waiting futilely for some work to process it.

The worse thing is that calling `try_lock` from a thread that already holds the mutex results in undefined behavior. If two text changes occur in rapid succession on the UI thread, the second one will try to lock the `m_workMutex` from the same thread that already locked it, and you have now broken the rules and anything could happen.

Even worse than the worse case is the possibility that the mutex is released from the wrong thread. This code switches back to the UI thread before allowing the `unique_lock` to destruct, so you think you're safe, but you're not because an exception while building the matches will result in the lock being destructed from a background thread. This happens before the promise's `unhandled_exception` is called, so you've corrupted the system before your `fire_and_forget` can fail fast.

The `m_workMutex` is really a red herring. It doesn't need to be a mutex. The code uses it merely as a flag. So let's switch to a flag and avoid all the undefined behavior.

Also, the `m_textMutex` is unnecessary since the `m_pendingText` is always accessed from the UI thread, so there is no concurrency. We can get rid of that too.

We're now left with this:

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

    bool m_busy = false;
    // std::mutex m_textMutex; // no longer needed
    std::optional<string> m_pendingText;
};

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

    if (std::exchange(m_busy, true)) {
        m_pendingText = text;
        co_return;
    }

    while (true) {
        co_await winrt::resume_background();

        std::vector<std::string> matches;
        for (auto&& candidate : FindCandidates(text)) {
            if (candidate.Verify()) {
                matches.push_back(candidate.Text());
            }
        }

        co_await winrt::resume_foreground(Dispatcher());

        SetAutocomplete(matches);

        if (!m_pendingText) {
            m_busy = false;
            co_return;
        }
        text = std::move(*m_pendingText);
        m_pendingText.reset();
    }
}
```

We can simplify the code by simply treating every case as the pending case.

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

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

    while (m_pendingText) {
        text = std::move(*m_pendingText);
        m_pendingText.reset();

        co_await winrt::resume_background();

        std::vector<std::string> matches;
        for (auto&& candidate : FindCandidates(text)) {
            if (candidate.Verify()) {
                matches.push_back(candidate.Text());
            }
        }

        co_await winrt::resume_foreground(Dispatcher());

        SetAutocomplete(matches);

    }
    m_busy = false;
}
```

The UI thread is doing a lot of heavy lifting here because it implicitly locks the combined accesses to `m_busy` and `m_pendingText`.

Next time, we'll try to reduce the amount of unnecessary work.