Being more adamant about reporting that C++/WinRT was unable to resume execution on a dispatcher thread

devblogs.microsoft.com/oldnewthing/20250722-00/?p=111400

July 22, 2025

Last time, we saw what happens if C++/WinRT is unable to resume execution on a dispatcher thread: If you use `winrt::resume_foreground` with a `CoreDispatcher`, then the coroutine simply never resumes (which means that it appears to have hung) and leaks. If you use `winrt::resume_foreground` with a `DispatcherQueue`, then the result of the `co_await` tells you whether the thread switch was successful, but in practice nobody actually checks the result.

Both of these are just bad situations to be in. In the `DispatcherQueue` case, you have to remember to check a value that in practice nobody ever checks. In the `CoreDispatcher` case, there's simply nothing you can do at all.

The Windows Implementation Library provides an alternative.

We get rid of the pit of failure. None of this "check a value that is easy to overlook". If the coroutine cannot resume on the dispatcher thread, it throws an exception, specifically `HRESULT_FROM_WIN32(ERROR_NO_TASK_QUEUE)`. It's hard to ignore an exception. You need to write special "exception-ignoring" code. And as a rule, C++/WinRT uses exceptions to report that things have gone wrong, so this is consistent with the rest of the C++/WinRT library.

Now we can complete the table of behaviors if you call `resume_foreground` and the dispatcher is unable to accept the work item.

	winrt::resume_foreground	wil::resume_foreground
CoreDispatcher	hang	throw "no task queue"
DispatcherQueue	return false	throw "no task queue"

One thing to note is that the "no task queue" exception is thrown from an arbitrary thread. The original thread is no longer available, since we suspended on it, and the thread is now doing something else. And the destination thread is not available because that's why we're throwing the exception. We're stuck in a no-man's land where nobody has access

to a thread, so the exception is just thrown from wherever it can. If you need to run destructors on or handle the exception from a specific thread, you'll have to switch to that thread yourself.

Bonus chatter: How does `wil::resume_foreground()` work?

The client-provided delegate is wrapped inside another delegate which keeps track of whether it has ever been called. If the delegate finds itself destructed without being called,[1] then it knows that the thread switch failed, and it sets `orphaned` to `true` before manually resuming the coroutine. The awaiter's `await_resume()` function does the work of throwing the "no task queue" exception if the `orphaned` flag is set.

[1] There's an edge case here: If the dispatcher rejects the delegate with an exception, then propagate that exception and don't throw another exception for an uncalled delegate.