An ignored exception can be misinterpreted as a hang, particularly in coroutines

devblogs.microsoft.com/oldnewthing/20230406-00

April 6, 2023

Raymond Chen

Consider the following function:

```
void DoAwesomeThings()
{
    try {
        Step1();
        Step2();
        Step3();
    }
    catch (...)
    {
    }
}
```

If an exception occurs in `Step2()`, then from the point of view of `DoAwesomeThings()`, it will appear to have hung, since control never returned.

```
void DoAwesomeThings()
{
    try {
        Step1();
        printf("About to call Step2!\n");
        Step2();
        printf("Step2 returned!\n"); // never executes!
        Step3();
    }
    catch (...)
    {
    }
}
```

Now, one difference is that if you break into the debugger, you'll find that there is no call to `Step2()` anywhere on the stack, and that may be a clue that the function didn't actually hang. But if all you have is log file, your log file just ends at

```
About to call Step2!
```

and if you didn't think about the `catch (...)`, you would conclude that `Step2()` is hung.

Okay, so that was pretty obvious. I mean, who would make this kind of misinterpretation?

Let's make it a little less obvious: Let's put it in a coroutine.

```
winrt::IAsyncAction DoAwesomeThings()
{
    Step1();
    Step2();
    Step3();
}
```

Recall that the coroutine transformation, among other things, wraps the entire coroutine body inside a `try...catch`:

```
winrt::IAsyncAction DoAwesomeThings()
{
    co_await promise.initial_suspend();
    try {
        Step1();
        printf("About to call Step2!\n");
        Step2();
        printf("Step2 returned!\n"); // never executes!
        Step3();
    } catch (...) {
        promise.unhandled_exception();
    }
    co_await promise.final_suspend();
}
```

This time, when `Step2()` throws an exception, it is caught by the coroutine infrastructure which calls the promise's `unhandled_exception()`. The `unhandled_exception()` function typically saves the exception somewhere, so that it can be rethrown when somebody does a `co_await` on the coroutine return object.

In this case, what happens is that the exception is saved in the `IAsyncAction` object, ready to be re-thrown by the `co_await` in `co_await DoAwesomeThings()`.

But say you never do that `co_await`.

```
void BeReallyAwesome()
{
    DoAwesomeThings(); // no co_await
    DoMoreAwesomeThings();
}
```

If you look at the log file, you see the "About to call Step2!" and no "Step2 returned!". And since you never did a `co_await`, the exception that was saved in the `IAsyncAction` is discarded without ever having a chance to be rethrown. Result: It looks like the coroutine

simply hung.

Note that each coroutine return type makes its own decision about how unhandled exceptions are dealt with, and what to do if an exception never gets rethrown. Here's a table of some of the ones you're like to encounter in Windows code.

Coroutine type	Unhandled exception	`co_await`	Unobserved exception
`winrt::IAsync...`	Save for later	Rethrow	Discarded
`winrt::fire_and_forget`	Fail fast	N/A	N/A
`Concurrency::task`	Save for later	Rethrow	Fail fast

For the last two popular Windows coroutine types, the exception gets reported eventually. The `fire_and_forget` crashes the process immediately, and `Concurrency::task` rethrows the exception (if observed) or crashes the process (if never observed).

It's the `IAsync...` that can cause exceptions to mysteriously vanish: If you simply discard them without every performing a `co_await`, then you never learn of any unhandled exceptions that occurred.

They just disappear.

We'll apply this knowledge next time.