How to wait for multiple C++ coroutines to complete before propagating failure, concluding remarks

devblogs.microsoft.com/oldnewthing/20230710-00

July 10, 2023

Raymond Chen

For the past several articles, we've looked at how we could write a `when_all_completed` function that awaits a set of awaitable objects, and only after all of them have been awaited does it propagate any exception that occurred. A lot of our work was erasing sources of exceptions that could occur *before* all of the awaitables have completed, most notable "out of memory" exceptions in the internal bookkeeping. We eventually were able to get rid of all the dynamic memory allocations in the coroutine body.

So we're done, right?

Well, no, because things could go wrong before we get to the coroutine body.

Consider this usage:

```
co_await when_all_completed(First(), Second(), Third());
```

This breaks down into multiple steps:

```
auto first = First();
auto second = Second();
auto third = Third();
auto action = when_all_completed(first, second, third);
co_await action;
```

(I've removed some `std::move()` calls for expository simplicity.)

All of our work to get rid of exceptions in the body of `when_all_completed` ensure that the `co_await action` throws an exception only because one of the awaitable parameters threw an exception, and not because we encountered a `std::bad_alloc` trying to do our work.

However, there are all the other steps that lead to the `co_await` that could still go wrong.

For example, if we make it past the calls of `First()` and `Second()`, but the call to `Third()` throws an exception, then the exception will propagate out of the full expression, and the `First()` and `Second()` coroutines will be left running without having ever been awaited.

So it's not good enough to say

```
try {
    co_await when_all_completed(First(), Second(), Third());
} catch (...) {
    /* Ignore errors, but at least we know that all of
     * the coroutines have completed, right? */
}
```

If an exception occurs while producing the coroutines, then that exception is thrown, and `when_all_completed` is never even called. Any coroutines which started prior to the exception are still running.[1]

There's also the possibility of a `std::bad_alloc` thrown by the failed allocation of the coroutine frame. We were able to eliminate this in our custom promise by preallocating space and using a secret signal to pass this preallocated space to the promise's `operator new`. However, we want `when_all_completed` to return a C++/WinRT `IAsyncAction`, and we don't control the `operator new` of the associated promise; that one came from C++/WinRT.

The best we can do is to catch the exception from the creation of the `IAsyncAction` and fail fast.

```
template<typename... T>
IAsyncAction when_all_complete_worker(T... asyncs)
{
    std::exception_ptr eptr;

    auto accumulate = [&](std::exception_ptr e) {
        if (eptr == nullptr) eptr = e;
    };

    (accumulate(co_await wrapped_awaitable(asyncs)), ...);

    if (eptr) std::rethrow_exception(eptr);
}

template<typename... T>
IAsyncAction when_all_completed(T&&... async) noexcept
{
    return when_all_complete_worker(
        std::forward<T>(async)...);
}

fire_and_forget SomeFunction()
{
    auto first = First();
    auto second = Second();
    auto third = Third();
    try {
        co_await when_all_completed(
            std::move(first),
            std::move(second),
            std::move(third));
    } catch (...) {
        LOG_CAUGHT_EXCEPTION();
    }
}
```

We move the coroutine into a helper function `when_all_complete_worker()` and and the new `when_all_complete()` function just forwards the parameters to the worker, but does so in a `noexcept` function, so if there is a problem allocating the coroutine frame or copying the parameters into the coroutine frame, the exception triggers a `std::terminate`.

Note that we called `First()`, `Second()`, and `Third()` outside of the `try`. This means that if (say) `Third()` throws an exception without producing an awaitable, the exception propagates immediately, and the coroutines `First()` and `Second()` are never awaited to completion.

That's probably not what you want, since the point of running them to completion is to make sure they have stopped doing work before allowing the next thing to happen.

So the calls to create the three coroutines should probably be inside a `noexcept`.

```
// Go back to the old when_all_completed

fire_and_forget SomeFunction()
{
    try {
        co_await [&amp;]() noexcept {
            return when_all_completed(
                First(),
                Second(),
                Third());
        }();
    } catch (...) {
        LOG_CAUGHT_EXCEPTION();
    }
}
```

This is still awkward enough that it's unlikely anybody would write it.

How about changing `when_all_completed` so it returns the exception instead of rethrowing it?

```
template<typename... T>
com_aware_task<std::exception_ptr>
when_all_complete(T... asyncs)
{
    std::exception_ptr eptr;

    auto accumulate = [&](std::exception_ptr e) {
        if (eptr == nullptr) eptr = e;
    };

    (accumulate(co_await wrapped_awaitable(asyncs)), ...);

    co_return eptr;
}

fire_and_forget SomeFunction()
{
    auto eptr = co_await when_all_completed(
            First(), Second(), Third());
    if (eptr) {
        try {
            std::rethrow_exception(eptr);
        } catch (...) {
            LOG_CAUGHT_EXCEPTION();
        }
    }
}
```

This time, we allow any exceptions from `First()`, `Second()`, `Third()`, or `when_all_completed` itself to propagate outward, which in the case of a `winrt::fire_and_forget` means immediate termination. If any of the awaitables complete with an exception, we log the exception (by rethrowing it and immediately catching it with a logging function). If this is pattern you intend to use a lot, you could wrap it in a helper.

```
void LogExceptionPtr(std::exception_ptr eptr)
{
    if (eptr) try {
        std::rethrow_exception(eptr);
    } CATCH_LOG();
}

fire_and_forget SomeFunction()
{
    LogExceptionPtr(co_await when_all_completed(
            First(), Second(), Third()));
}
```

But if you're going to be logging and discarding exceptions, maybe you want to just make `when_all_completed` do it. After all, there could be multiple exceptions, and `when_all_completed` returns only the first one.

```
template<typename... T>
IAsyncAction when_all_complete(T... asyncs)
{
    std::exception_ptr eptr;

    auto accumulate = [&](std::exception_ptr e) {
        if (e) {
            LogExceptionPtr(e);
            if (eptr == nullptr) eptr = e;
        }
    };

    (accumulate(co_await wrapped_awaitable(asyncs)), ...);

    if (eptr) std::rethrow_exception(eptr);
}
```

That said, you may want to make `LogExceptionPtr` a macro, so that the error log will have a line number that points to the `LOG_EXCEPTION_PTR` rather than to the helper function.

```
#define LOG_EXCEPTION_PTR(eptr) \
do { if (auto&& __eptr = eptr) \
    try { std::rethrow_exception(__eptr); } CATCH_LOG(); } while (0)
```

After all this discussion, maybe your conclusion is that `when_all_completed` is just too weird, because it conflates exceptions that occur while producing or consuming the awaitables with exceptions that are produced by the awaitables themselves. And the correct answer is "Don't

use it. Just catch the exceptions in the awaitables themselves.”

```
IAsyncAction First() try
{
    /* original body of the First() function */
} CATCH_LOG();

IAsyncAction Second() try
{
    /* original body of the Second() function */
} CATCH_LOG();

IAsyncAction Third() try
{
    /* original body of the Third() function */
} CATCH_LOG();

fire_and_forget SomeFunction()
{
    co_await when_all(
            First(), Second(), Third()));
}
```

We caught the exceptions in the awaitables and ignored them (after logging). This means that `when_all()` doesn't have to report any exceptions in the awaitables, since we removed them. Any exceptions that come out of the line `co_await when_all(First(), Second(), Third())` were the result either of producing or consuming the awaitables, and those are probably failures you want to treat as fatal errors since you have no reasonable way of recovering from them.

There, I managed to convince myself that, after spending a very long time trying to write the `when_all_complete` function, it was a bad idea after all.

[1] The Windows Runtime uses eager-start coroutines, so the coroutine starts on creation. If you use lazy-start coroutines, then the exception is thrown even before the coroutine starts, and you end up abandoning the coroutine without starting it. You don't have the problem of a coroutine still running, but you do have the problem of a coroutine that never started.