Inside C++/WinRT: Apartment switching: Unblocking the outgoing thread

devblogs.microsoft.com/oldnewthing/20230125-00

January 25, 2023

Raymond Chen

Last time, we built an `apartment_context` object and used it as part of our coroutine infrastructure so that `co_await` of Windows Runtime asynchronous operations resume in the same COM context as they started.

Our implementation of the `apartment_context` suffers from the problem of the synchronous apartment-changing callback: The thread being switched *from* waits synchronously for the thread being switched *to*. This is a bad thing if the thread being switched *from* is a UI thread that needs to keep the UI responsive, whereas the thread being switched *to* is a background thread that is happy to make long blocking calls.

C++/WinRT addresses this problem by hopping through a background thread in cases where it thinks this will be a problem. In order to detect whether we are in a problem case, we use the `CoGetApartmentType` function to determine what kind of apartment we are in.

```
inline std::pair<int32_t, int32_t> get_apartment_type() noexcept
{
    int32_t aptType;
    int32_t aptTypeQualifier;
    check_hresult(WINRT_IMPL_CoGetApartmentType(&aptType, &aptTypeQualifier));
    return { aptType, aptTypeQualifier };
}

inline bool is_sta_thread() noexcept
{
    auto type = get_apartment_type();
    switch (type.first)
    {
    case 0: /* APTTYPE_STA */
    case 3: /* APTTYPE_MAINSTA */
        return true;
    case 2: /* APTTYPE_NA */
        return type.second == 3 /* APTTYPEQUALIFIER_NA_ON_STA */ ||
            type.second == 5 /* APTTYPEQUALIFIER_NA_ON_MAINSTA */;
    }
    return false;
}
```

The code that switches apartments now checks whether it is on a thread that hosts a single-threaded apartment (STA). If so, then instead of resuming the coroutine synchronously via `ContextCallback()`, it schedules the work to a background thread. That way, it is a background thread that blocks on a potentially long-running coroutine rather than a UI thread.

First, we rename our old `resume_apartment` function so it represents the *synchronous* resumption of the coroutine in another apartment.

```
void resume_apartment_sync(
    com_ptr<IContextCallback> const& context,
    std::coroutine_handle<> handle)
{
    com_callback_args args{};
    args.data = handle.address();

    check_hresult(
        context->ContextCallback(resume_apartment_callback,
            &args,
            guid_of<ICallbackWithNoReentrancyToApplicationSTA>(),
            5, nullptr));
}
```

And then we write a new `resume_apartment` that resumes either synchronously or asynchronously, depending on the apartment context.

```
inline auto resume_apartment(
    com_ptr<IContextCallback> const& context,
    coroutine_handle<> handle)
{
    WINRT_ASSERT(context.valid());
    if (is_sta_thread())
    {
        resume_apartment_on_threadpool(context, handle);
    }
    else
    {
        resume_apartment_sync(context, handle);
    }
}
```

To resume a coroutine in an apartment from the threadpool, we do it in two steps. First, schedule work on the threadpool. The work consists of synchronously switching to the destination apartment for resuming the coroutine.

```
struct threadpool_resume
{
    threadpool_resume(com_ptr<IContextCallback> const& context,
        coroutine_handle<> handle) :
        m_context(context), m_handle(handle) { }
    com_ptr<IContextCallback> m_context;
    coroutine_handle<> m_handle;
};

inline void __stdcall fallback_submit_threadpool_callback(
    void*, void* p) noexcept
{
    std::unique_ptr<threadpool_resume>
        state{ static_cast<threadpool_resume*>(p) };
    resume_apartment_sync(state->m_context, state->m_handle);
}

inline void resume_apartment_on_threadpool(
    com_ptr<IContextCallback> const& context,
    coroutine_handle<> handle)
{
    auto state = std::make_unique<threadpool_resume>(context, handle);
    submit_threadpool_callback(fallback_submit_threadpool_callback,
                               state.get());
    state.release();
}
```

The `resume_apartment_on_threadpool` function captures its parameters into a `threadpool_resume` structure and uses that pointer as the context pointer for a threadpool callback. The threadpool callback reconstitutes the `unique_ptr` and uses `resume_apartment_sync` to resume the coroutine inside the apartment.

We have avoided the problem of the synchronous apartment-changing callback blocking a UI thread for an extended period of time. We detected the dangerous situation and moved the work to a threadpool thread, which is not a UI thread.

Next time, we'll fix another problem with this implementation.