cppcon

Putting Coroutines to Work with the Windows Runtime

Kenny Kerr

Microsoft Windows

@KennyKerr

James McNellis

Microsoft Visual C++

@JamesMcNellis

```
std::future<int> ProduceAsync()
{
    return std::async(std::launch::async, []
    {
        // deep thought...
        return 42;
    });
}
```

Producing async today

```
void ConsumeAsync()
{
    printf("waiting\n");

    int answer = ProduceAsync().get();

    printf("done %d\n", answer);
}
```

Consuming async today

```
std::future<int> ProduceAsync();

std::future<void> ConsumeAsync()
{
    printf("waiting\n");

    return std::async(std::launch::async, []
    {
        int answer = ProduceAsync().get();

        printf("done %d\n", answer);
    });
}
```

Continuation today

```
task<String ^> OcrAsync(String ^ path)
{
    return create_task(StorageFile::GetFileFromPathAsync(path))
        .then([](StorageFile ^ file) {
        return file->OpenAsync(FileAccessMode::Read);
    })
        .then([](IRandomAccessStream ^ stream) {
        return BitmapDecoder::CreateAsync(stream);
    })
        .then([](BitmapDecoder ^ decoder) {
        return decoder->GetSoftwareBitmapAsync();
    })
        .then([](SoftwareBitmap ^ bitmap) {
        OcrEngine ^ engine = OcrEngine::TryCreateFromUserProfileLanguages();
        return engine->RecognizeAsync(bitmap);
    })
        .then([](OcrResult ^ result) {
        return result->Text;
    });
}
```

You really want future::then?

```
std::future<hstring> AsyncSample(hstring_ref path)
{
    StorageFile file = co_await StorageFile::GetFileFromPathAsync(path);

    IRandomAccessStream stream = co_await file.OpenAsync(FileAccessMode::Read);

    BitmapDecoder decoder = co_await BitmapDecoder::CreateAsync(stream);

    SoftwareBitmap bitmap = co_await decoder.GetSoftwareBitmapAsync();

    OcrEngine engine = OcrEngine::TryCreateFromUserProfileLanguages();

    OcrResult result = co_await engine.RecognizeAsync(bitmap);

    return result.Text();
}
```

Async with C++/WinRT

```
std::future<int> ProduceAsync();

std::future<void> ConsumeAsync()
{
    printf("waiting\n");

    int answer = co_await ProduceAsync();

    printf("done %d\n", answer);
}
```

Consumption & continuation with co_await

```
concurrency::task<int> ProduceAsync();

concurrency::task<void> ConsumeAsync()
{
    printf("waiting\n");

    int answer = co_await ProduceAsync();

    printf("done %d\n", answer);
}
```

std::future or concurrency::task?

```
IAsyncOperation<int> ProduceAsync();

IAsyncAction ConsumeAsync()
{
    printf("waiting\n");

    int answer = co_await ProduceAsync();

    printf("done %d\n", answer);
}
```

Say hello to C++/WinRT async

Blocking

```
IAsyncOperation<int> Produce();

void Consume()
{
    int result = Produce().get();
    printf("%d\n", result);
}
```

Blocking wait

```
template <typename TResult>
struct IAsyncOperation : IAsyncInfo
{
    using completed_handler = AsyncOperationCompletedHandler<TResult>;

    void Completed(completed_handler const & handler) const;
    completed_handler Completed() const;
    TResult GetResults() const;

    TResult get() const
    {
        // blocking suspend...
        return GetResults();
    }
};
```

.get

```
template <typename Async> void blocking_suspend(const Async & async)
{
    if (async.Status() == AsyncStatus::Completed)
    {
        return;
    }

    lock x;
    condition_variable cv;
    bool completed = false;

    async.Completed([&](Async const &, AsyncStatus)
    {
        winrt::lock_guard const guard(x);
        completed = true;
        cv.wake_one();
    });

    winrt::lock_guard const guard(x);
    cv.wait_while(x, [&] { return !completed; });
}
```

Blocking suspend

```
IAsyncOperation<int> Produce();

void Consume()
{
    int result = Produce().get();
    printf("%d\n", result);
}
```

From blocking wait…

```
IAsyncOperation<int> Produce();

IAsyncAction Consume()
{
    int result = co_await Produce();
    printf("%d\n", result);
}
```

To co_await

Async with C++/WinRT

```
struct IAsyncInfo : IInspectable
{
    virtual HRESULT get_Id(uint32_t * id) = 0;
    virtual HRESULT get_Status(AsyncStatus * status) = 0;
    virtual HRESULT get_ErrorCode(HRESULT * errorCode) = 0;
    virtual HRESULT Cancel() = 0;
    virtual HRESULT Close() = 0;
};


IAsyncInfo * info = ...

AsyncStatus status;
HRESULT hr = info->get_Status(&status);

info->Release();
```

WinRT as COM

```
struct IAsyncInfo : IInspectable
{
    uint32_t Id() const;
    AsyncStatus Status() const;
    HRESULT ErrorCode() const;
    void Cancel() const;
    void Close() const;
};


IAsyncInfo info = ...

AsyncStatus status = info.Status();
```

C++/WinRT

Actions and Operations

```
IAsyncAction
◦ void    Completed(handler)
◦ handler Completed()
◦ void    GetResults()
```

```
IAsyncOperation<T>
◦ void    Completed(handler)
◦ handler Completed()
◦ T       GetResults()
```

```
IAsyncActionWithProgress<P> adds
◦ void    Progress(handler)
◦ handler Progress()
```

```
IAsyncOperationWithProgress<T, P> adds
◦ void    Progress(handler)
◦ handler Progress()
```

```
using namespace Windows::Storage;

StorageFolder folder = KnownFolders::VideosLibrary();


StorageFile file = co_await folder.GetFileAsync(L"Before.mp4");


co_await file.RenameAsync(L"After.mp4");
```

IAsyncOperation<StorageFile>

IAsyncAction

IAsyncOperation & IAsyncAction

```
IAsyncAction Sample(SyndicationClient const & client, Uri const & uri)
{
    SyndicationFeed feed = co_await client.RetrieveFeedAsync(uri);

    for (SyndicationItem item : feed.Items())
    {
        hstring title = item.Title().Text();
        printf("%ls\n", title.c_str());
    }
}
```

IAsyncOperationWithProgress
<SyndicationFeed, RetrievalProgress>

Progress

```
IAsyncAction Sample(SyndicationClient const & client, Uri const & uri)
{
    // SyndicationFeed feed = co_await client.RetrieveFeedAsync(uri);

    auto async = client.RetrieveFeedAsync(uri);

    async.Progress([](auto const & sender,
                      RetrievalProgress const & progress)
    {
        printf("Received %d of %d bytes\n",
               progress.BytesRetrieved,
               progress.TotalBytesToRetrieve);
    });

    SyndicationFeed feed = co_await async;

    for (SyndicationItem item : feed.Items())
    {
        hstring title = item.Title().Text();
        printf("%ls\n", title.c_str());
    }
}
```

1. Make the call

2. Sign up

```
Received 32768 of 78158 bytes
Received 65536 of 78158 bytes
Received 78158 of 78158 bytes

Microsoft + Modern
Available on GitHub
Clang and the Windows Runtime
Universal Windows Apps with Standard C++
When Standard C++ Isn't Enough
Modern C++ as a Better Compiler
A Classy Type System for Modern C++
Coming Soon
```

```
struct RetrievalProgress
{
    uint32_t BytesRetrieved;
    uint32_t TotalBytesToRetrieve;
};

IAsyncOperationWithProgress<SyndicationFeed, RetrievalProgress>
    RetrieveFeedAsync(Uri const & uri)
{
    auto report_progress = co_await get_progress_token;

    for (unsigned i = 1; i <= 10; ++i)
    {
        // download next chunk...
        report_progress({ i * 10, 100 });
    }

    SyndicationFeed feed = ...
    return feed;
}
```

Reporting progress

```
IAsyncAction ChangeTheWorldAsync()
{
    auto canceled = co_await get_cancellation_token;
    for (unsigned i = 1; i <= 10; ++i)
    {
        printf("Change...\n");
        co_await 1s;

        if (canceled())
        {
            printf("Canceled.\n");
            return;
        }
    }
}

IAsyncAction App()
{
    auto async = ChangeTheWorldAsync();
    co_await 2s;
    async.Cancel();
}
```

```
Change...
Change...
Canceled.
```

Supporting cancellation

Making WinRT Async Types Awaitable

```
int result = co_await Produce();
```

```
printf("%d\n", result);
```

co_await

```
// int result = co_await Produce();

IAsyncOperation<int> temp = Produce();

if (!temp.await_ready())
{
    temp.await_suspend(...);

    // Suspension point
}

int result = temp.await_resume();

printf("%d\n", result);
```

Option 1: awaitable type

```
template <typename Async>
struct await_adapter
{
    Async const & async;

    bool await_ready() const;
    void await_suspend(std::experimental::coroutine_handle<> handle) const;
    auto await_resume() const;
};

template <typename T>
await_adapter<IAsyncOperation<T>> operator co_await(IAsyncOperation<T> const & async)
{
    return{ async };
}
```

Option 2: operator co_await

```
bool await_ready() const
{
    return async.Status() == AsyncStatus::Completed;
}
```

Are you ready?

```
void await_suspend(coroutine_handle<> handle) const
{
    com_ptr<IContextCallback> context;
    check_hresult(CoGetObjectContext(...));

    async.Completed([handle, context = std::move(context)](...)
    {
        ComCallData data = {};
        data.pUserDefined = handle.address();

        check_hresult(context->ContextCallback([](ComCallData * data)
        {
            coroutine_handle<>::from_address(handle);
            return S_OK;
        }, &data, ...));
    });
}
```

Not ready? Let’s suspend…

```
auto await_resume() const
{
    return async.GetResults();
}
```

Resumed, get results

```
try
{
    int result = Produce().get();

    ...
}
catch (hresult_invalid_argument const & e)
{
    ...
}
```

Synchronous exception handling

```
try
{
    int result = co_await Produce();

    ...
}
catch (hresult_invalid_argument const & e)
{
    ...
}
```

Cooperative exception handling

Using WinRT Async Types as Coroutine Types

```
IAsyncAction Produce()
{
    co_await 1s;
}
```

Being a coroutine type

```
template <typename ... Args>
struct coroutine_traits<IAsyncAction, Args ...>
{
    struct promise_type
    {
        IAsyncAction get_return_object();

        void return_void();
        void set_exception(exception_ptr const &);

        suspend_never initial_suspend();
        suspend_never final_suspend();
    };
};
```

1. Create

3. Get result

5. Complete

2. Proceed?

6. Can destroy?

4. Run to suspension

Coroutine lifecycle

```
struct Async : implements<Async, IAsyncAction, IAsyncInfo>
{
    AsyncActionCompletedHandler m_handler;

    void Completed(AsyncActionCompletedHandler const & handler)
    { m_handler = m_handler; }

    AsyncActionCompletedHandler Completed()
    { return m_handler; }

    void GetResults() {}

    uint32_t Id() { return 1; }
    AsyncStatus Status() { return AsyncStatus::Started; }
    HRESULT ErrorCode() { return S_OK; }
    void Cancel() {}
    void Close() {}
};
```

IAsyncAction

IAsyncInfo

Implementing IAsync…

```
struct promise_type
{
    IAsyncAction m_async{ make<Async>(); };

    IAsyncAction get_return_object()
    {
        return m_async;
    }

    void return_void();
    void set_exception(exception_ptr const &);

    suspend_never initial_suspend();
    suspend_never final_suspend();
};
```

1. Create & hold reference

2. Return second reference

3. Safe to release first reference!

Coroutines and COM objects

```
struct promise_type
{
    IAsyncAction m_async{ make<Async>(); };

    IAsyncAction get_return_object()
    {
        return m_async;
    }

    void return_void();
    void set_exception(exception_ptr const &);

    suspend_never initial_suspend();
    suspend_never final_suspend();
};
```

1. First allocation

2. Second allocation

Allocations!

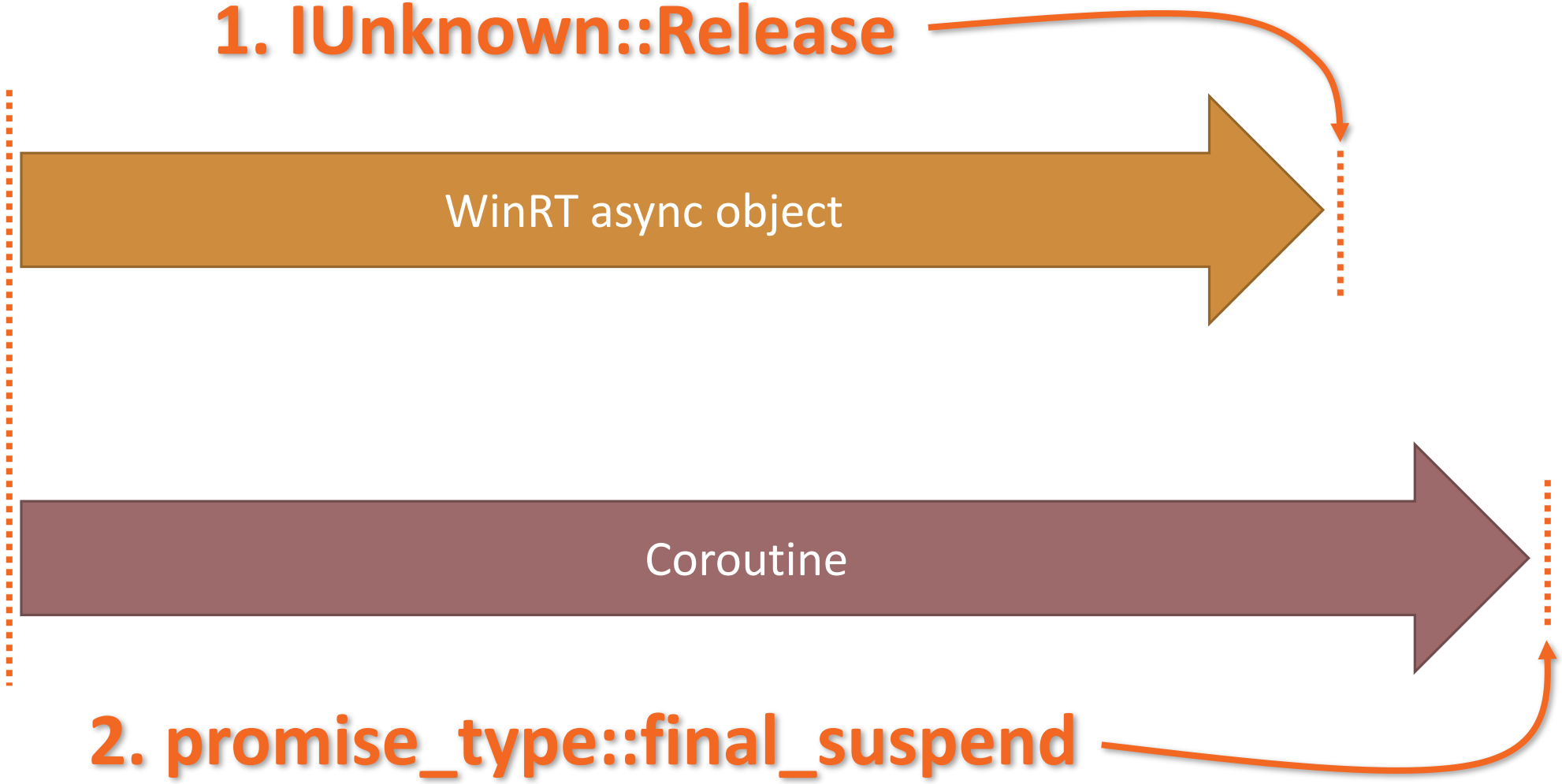
```
struct promise_type : implements<promise_type,
                                 IAsyncAction,
                                 IAsyncInfo>
{
    IAsyncAction get_return_object()
    {
        return *this;
    }

    ...

    AsyncStatus Status();

    ...
};
```

1. Single allocation

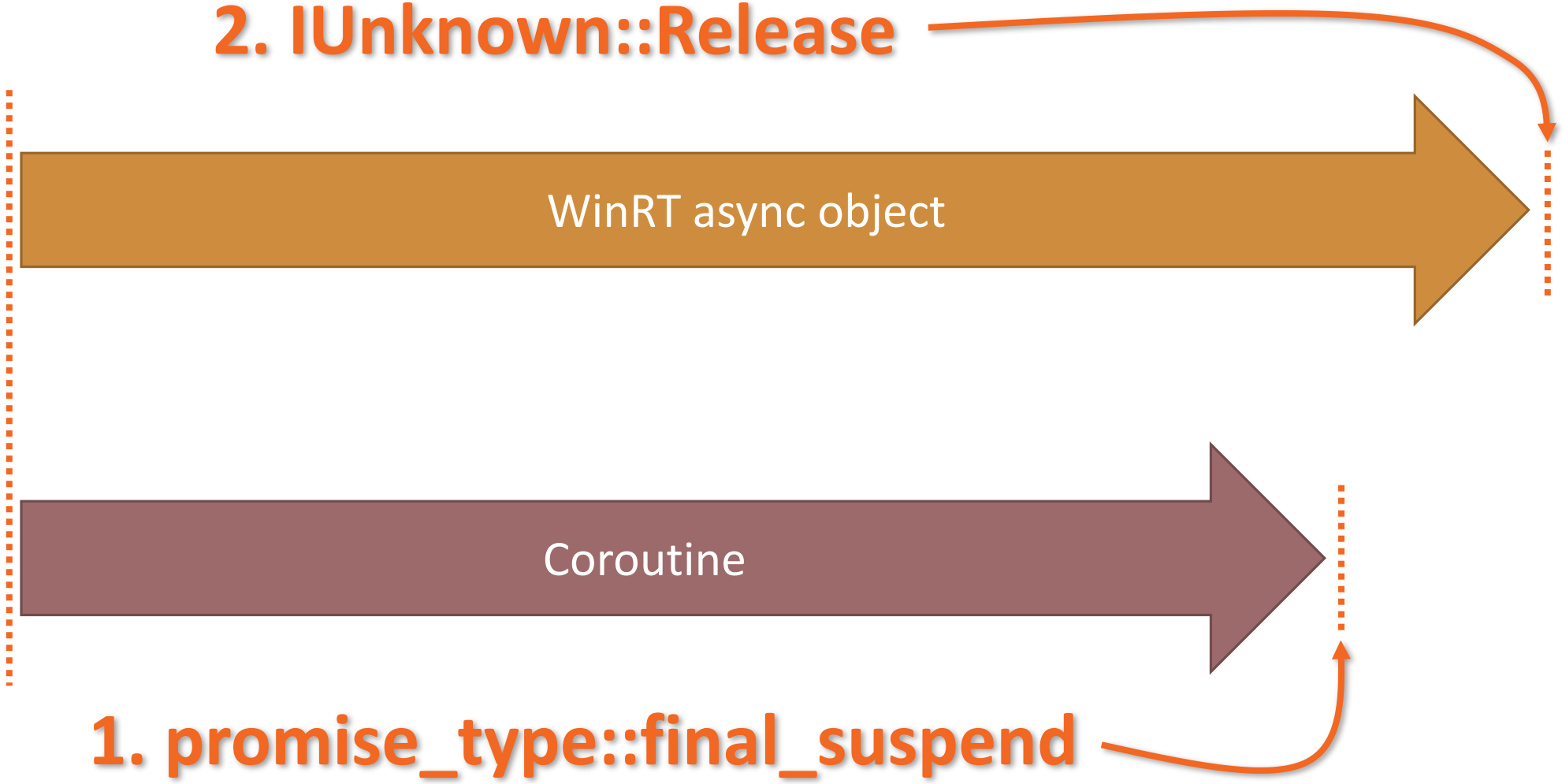
2. promise implementation

3. WinRT implementation

Coroutines as COM objects



Lifetime could be this…



… or lifetime could be this

This is called a condition race. :)

This is called a race condition. :)

```
atomic<uint32_t> m_references{ 1 };

unsigned long AddRef()
{
    return 1 + m_references.fetch_add(1, memory_order_relaxed);
}

unsigned long Release()
{
    uint32_t const remaining = m_references.fetch_sub(1, memory_order_release) - 1;

    if (remaining == 0)
    {
        atomic_thread_fence(memory_order_acquire);
        delete this;
    }

    return remaining;
}
```

Could be a problem!

COM reference counting

```
struct promise_type
{
    ...

    suspend_never final_suspend()
    {
        return{};
    }
};
```

Destroy coroutine automatically

This won't work!

promise_type lifetime

```
struct promise_type
{
    ...

    suspend_always final_suspend()
    {
        return{};
    }
};
```

Coroutine destroyed manually

This won't work either! :(

promise_type lifetime

```
struct promise_type : implements<promise_type, IAsyncAction, IAsyncInfo>
{
    uint32_t just_release() noexcept
    {
        return this->m_references.fetch_sub(1, memory_order_release) - 1;
    }

    unsigned long __stdcall Release() noexcept
    {
        uint32_t const remaining = just_release();

        if (remaining == 0)
        {
            atomic_thread_fence(memory_order_acquire);
            coroutine_handle<promise_type>::from_promise(*this).destroy();
        }

        return remaining;
    }

    ...
```

Override implements<…>

Destroy coroutine here

COM ♥ C++ coroutines

```
struct promise_type : implements<promise_type, IAsyncAction, IAsyncInfo>
{
    struct final_suspend_type
    {
        promise_type * promise;

        bool await_ready() { return false; }
        void await_resume() {}

        bool await_suspend(coroutine_handle<>)
        {
            return 0 < promise->just_release();
        }
    };

    final_suspend_type final_suspend() noexcept
    {
        return{ this };
    }

    ...
```

1. Release self-reference

2. Suspend or destroy?

C++ coroutines ♥ COM

Progress and Cancellation

```
IAsyncActionWithProgress<uint32_t> Produce()
{
    auto canceled        = co_await get_cancellation_token;
    auto report_progress = co_await get_progress_token;

    for (uint32_t i = 0; !canceled(); ++i)
    {
        co_await 1ms;

        report_progress(i);
    }
}
```

Supporting progress & cancellation

```
IAsyncAction Consume()
{
    auto async = Produce();

    async.Progress([](auto const & async, uint32_t value)
    {
        printf("%d\n", value);
    });

    co_await 3s;
    async.Cancel();
}
```

Using progress & cancellation

```
struct pass_through
{
    int result;

    bool await_ready() { return true; }
    void await_suspend(std::experimental::coroutine_handle<>) {}
    int await_resume() { return result; }
};

IAsyncAction Produce()
{
    int result = co_await pass_through{ 123 };
    printf("%d\n", result);
}
```

I'm blue!

Different faces of co_await: awaitable types

```
struct value
{
    int result;
};

pass_through operator co_await(value value)
{
    return{ value.result };
}

IAsyncAction Produce()
{
    int result = co_await value{ 123 };
    printf("%d\n", result);
}
```

I'm teal!

Different faces of co_await: operator co_await

```
struct promise_type : implements<promise_type, IAsyncAction, IAsyncInfo>
{
    template <typename Expression>
    Expression && await_transform(Expression && expression) noexcept
    {
        return forward<Expression>(expression);
    }

    ...
};
```

Different faces of co_await: await_transform

```
struct get_progress_token_t {};
constexpr get_progress_token_t get_progress_token {};

struct promise_type : implements<promise_type, IAsyncAction, IAsyncInfo>
{
    template <typename Expression>
    Expression && await_transform(Expression && expression) noexcept
    {
        return forward<Expression>(expression);
    }

    progress_type await_transform(get_progress_token_t) noexcept
    {
        return{ this };
    }

    ...
};
```

Access to promise/return object!

Transforming progress

```
struct progress_type
{
    promise_type * promise;

    bool await_ready() { return true; }
    void await_suspend(coroutine_handle<>) { }

    progress_type await_resume()
    {
        return *this;
    }

    void operator()(TProgress const & result)
    {
        promise->set_progress(result);
    }
};
```

Lightweight awaitable type

Never suspends

Returns function object

Notify listener

Awaitable progress_type

```
struct canceled_type
{
    promise_type * promise;

    bool await_ready() { return true; }
    void await_suspend(coroutine_handle<>) { }

    canceled_type await_resume()
    {
        return *this;
    }

    bool operator()()
    {
        return promise->Status() == async_status::Canceled;
    }
};
```

Have I been canceled?

Awaitable cancellation

Coroutines and The Thread Pool

```
IAsyncAction Produce()
{
}
```

error C4716: 'Produce': must return a value

This won’t compile

```
IAsyncAction Produce()
{
    co_await 1s;
}
```

Use existing awaitable type

Option 1

```
struct Async : implements<Async, IAsyncAction, IAsyncInfo>
{
    ...
};

IAsyncAction Produce()
{
    return make<Async>();
}
```

Forward an implementation

Option 2

Windows Thread Pool

Four kinds of submittable operations:

- Work
- Wait
- Timer
- I/O

Based on I/O completion ports

Thread Pool Work

```
IAsyncAction Produce()
{
    for (int y = 0; y < height; ++y)
    {
        for (int x = 0; x < width; ++x)
        {
            // Computationally expensive work...
        }
    }
}
```

?

Lacks a suspension point

Oh for a background thread

```
struct suspend_never
{
    bool await_ready() { return true; }
    void await_resume() {}
    void await_suspend(std::experimental::coroutine_handle<>) {}
};

IAsyncAction Produce()
{
    co_await suspend_never{};
    Sleep(5000);
}

int main()
{
    assert(Produce().Status() == AsyncStatus::Completed);
}
```

Awaitable but won't suspend

Valid but faulty coroutine

This blocks :(

Blocking coroutines

```
struct resume_background
{
    bool await_ready() { return false; }
    void await_resume() {}

    void await_suspend(coroutine_handle<> handle)
    {
        if (!TrySubmitThreadpoolCallback([](PTP_CALLBACK_INSTANCE, void * context)
        {
            coroutine_handle<>::from_address(context)();
        },
        handle.address(), nullptr))
        {
            throw_last_error();
        }
    }
};
```

1. Suspend

2. Submit threadpool work

3. Resume coroutine on thread pool

resume_background

```
IAsyncAction Produce()
{


    for (int y = 0; y < height; ++y)
    {
        for (int x = 0; x < width; ++x)
        {
            // Computationally expensive work...
        }
    }
}
```

Oh for a background thread

```
IAsyncAction Produce()
{
    co_await resume_background();

    for (int y = 0; y < height; ++y)
    {
        for (int x = 0; x < width; ++x)
        {
            // Computationally expensive work...
        }
    }
}
```

1. Calling thread context

2. Thread pool

Ah a background thread

```
IAsyncAction Produce()
{
    printf("(%d) Produce begin\n", GetCurrentThreadId());
    co_await suspend_never();
    Sleep(5000);
    printf("(%d) Produce end\n", GetCurrentThreadId());
}

int main()
{
    printf("(%d) main begin\n", GetCurrentThreadId());
    auto p1 = Produce();
    auto p2 = Produce();

    printf("(%d) main waiting\n", GetCurrentThreadId());
    p1.get();
    p2.get();

    printf("(%d) main end\n", GetCurrentThreadId());
}
```

Five sec. delay here…

And here…

Lack of concurrency

```
IAsyncAction Produce()
{
    printf("(%d) Produce begin\n", GetCurrentThreadId());
    co_await resume_background();
    Sleep(5000);
    printf("(%d) Produce end\n", GetCurrentThreadId());
}

int main()
{
    printf("(%d) main begin\n", GetCurrentThreadId());
    auto p1 = Produce();
    auto p2 = Produce();

    printf("(%d) main waiting\n", GetCurrentThreadId());
    p1.get();
    p2.get();

    printf("(%d) main end\n", GetCurrentThreadId());
}
```

Combined delay here

Lots of concurrency

no_suspend

(2712) main begin
(2712) Produce begin
← five second delay here
(2712) Produce end
(2712) Produce begin
← five second delay here
(2712) Produce end
(2712) main waiting
(2712) main end

resume_background

(11048) main begin
(11048) Produce begin
(11048) Produce begin
(11048) main waiting
← five second delay here

(3376) Produce end
(14244) Produce end
(11048) main end

```
IAsyncAction ForegroundAsync(TextBlock block)
{
    FileOpenPicker picker = ...
    auto file = co_await picker.PickSingleFileAsync();

    block.Text(co_await BackgroundAsync(file));
}

IAsyncOperation<hstring> BackgroundAsync(StorageFile file)
{
    co_await resume_background();

    auto stream = co_await file.OpenAsync(FileAccessMode::Read);
    ...
    auto result = co_await engine.RecognizeAsync(bitmap);
    return result.Text();
}
```

1. Requires UI thread

3. Back on UI thread

2. Offload work

Offloading work with “background” coroutine

```
struct thread_context
{
    thread_context()
    {
        check_hresult(CoGetObjectContext(... (put(m_context))));
    }
    bool await_ready() { return false; }
    void await_resume() {}
    void await_suspend(coroutine_handle<> handle)
    {
        ComCallData data = ...
        check_hresult(m_context->ContextCallback([](ComCallData * data)
        {
            coroutine_handle<>::from_address(data->pUserDefined)();
            return S_OK;
        }, ...
    }

    com_ptr<IContextCallback> m_context;
};
```

1. Get calling context

2. Resume on original context

Awaitable thread context

```
IAsyncAction Async(TextBlock block)
{
    FileOpenPicker picker = ...
    auto file = co_await picker.PickSingleFileAsync();

    thread_context ui_thread;

    co_await resume_background();

    auto stream = co_await file.OpenAsync(FileAccessMode::Read);
    ...
    auto result = co_await engine.RecognizeAsync(bitmap);

    co_await ui_thread;

    block.Text(result.Text());
}
```

1. On UI thread

2. Capture UI context

3. Switch to thread pool

4. Return to UI thread

Just one coroutine

Thread Pool Wait

```
IAsyncAction Produce(HANDLE event)
{
    co_await resume_background();
    // Computationally expensive work...
    SetEvent(event);
}

IAsyncAction Consume(HANDLE event)
{
    co_await resume_background();
    WaitForSingleObject(event, INFINITE);
    // Proceed...
}

int main()
{
    HANDLE event = CreateEvent(nullptr, true, false, nullptr);
    auto p = Produce(event);
    auto c = Consume(event);
    ...
}
```

Inefficient wait

```
struct resume_on_signal
{
    explicit resume_on_signal(HANDLE handle);
    resume_on_signal(HANDLE handle, TimeSpan timeout);

    bool await_ready() const noexcept
    {
        return WaitForSingleObject(m_handle, 0) == WAIT_OBJECT_0;
    }

    void await_suspend(std::experimental::coroutine_handle<> resume)
    {
        m_resume = resume;
        m_wait = CreateThreadpoolWait(callback, this, nullptr);
        SetThreadpoolWait(...);
    }

    bool await_resume() const noexcept
    {
        return m_result == WAIT_OBJECT_0;
    }

    ...
```

resume_on_signal

```
IAsyncAction Produce(HANDLE event)
{
    co_await resume_background();
    // Computationally expensive work...
    SetEvent(event);
}

IAsyncAction Consume(HANDLE event)
{
    co_await resume_background();
    WaitForSingleObject(event, INFINITE);
    // Proceed...
}

int main()
{
    HANDLE event = CreateEvent(nullptr, true, false, nullptr);
    auto p = Produce(event);
    auto c = Consume(event);
    ...
}
```

Inefficient wait

```
IAsyncAction Produce(HANDLE event)
{
    co_await resume_background();
    // Computationally expensive work...
    SetEvent(event);
}

IAsyncAction Consume(HANDLE event)
{
    co_await resume_on_signal(event);

    // Proceed...
}

int main()
{
    HANDLE event = CreateEvent(nullptr, true, false, nullptr);
    auto p = Produce(event);
    auto c = Consume(event);
    ...
}
```

Efficient wait

```
IAsyncAction Consume(HANDLE event)
{
    if (co_await resume_on_signal(event, 500ms))
    {
        // Proceed...
    }
    else
    {
        // Oh well...
    }
}
```

Wait with timeout

Thread Pool Timer

```
IAsyncAction Produce()
{
    co_await resume_background();

    for (uint32_t i = 1; i <= 3; ++i)
    {
        Sleep(i * 1000);

        // Retry after increasing delay...
    }
}
```

Inefficient timer

```
struct resume_after
{
    resume_after(Windows::Foundation::TimeSpan duration);

    bool await_ready()
    {
        return m_duration.count() <= 0;
    }

    void await_suspend(std::experimental::coroutine_handle<> handle)
    {
        m_timer = CreateThreadpoolTimer(callback, handle.address(), nullptr);
        SetThreadpoolTimer(...);
    }

    void await_resume() {}

    ...
};
```

resume_after

```
IAsyncAction Produce()
{
    co_await resume_background();

    for (uint32_t i = 1; i <= 3; ++i)
    {
        Sleep(i * 1000);

        // Retry after increasing delay...
    }
}
```

Inefficient timer

```
IAsyncAction Produce()
{


    for (uint32_t i = 1; i <= 3; ++i)
    {
        co_await resume_after(std::chrono::seconds(i));

         // Retry after increasing delay...
    }
}
```

Efficient timer

```
auto operator co_await(Windows::Foundation::TimeSpan duration)
{
    return resume_after(duration);
}


IAsyncAction Produce()
{
    printf("1s\n");
    co_await 1s;

    printf("500ms\n");
    co_await 500ms;

    printf("done!\n");
}
```

Making durations resumable

Thread Pool I/O

```
IAsyncAction ReadAsync()
{
    file reader(L"C:\\filename.txt");
    std::array<char, 1024> buffer;
    uint64_t offset = 0;

    while (uint32_t bytes_copied = await reader.read(offset, buffer.data(), buffer.size()))
    {
        printf("%.*s", bytes_copied, buffer.data());
        offset += bytes_copied;
    }
}
```

C++ Coroutines ❤ overlapped I/O

```
auto read(const uint64_t offset, void * const buffer, const size_t size)
{
    return m_io.start([=, handle = get(m_handle)](OVERLAPPED & overlapped)
    {
        overlapped.Offset = static_cast<DWORD>(offset);
        overlapped.OffsetHigh = offset >> 32;

        if (!ReadFile(handle, buffer, static_cast<DWORD>(size), nullptr, &overlapped))
        {
            const DWORD error = GetLastError();

            if (error != ERROR_IO_PENDING)
            {
                throw hresult_error(HRESULT_FROM_WIN32(error));
            }
        }
    });
}
```

C++ Coroutines ❤ overlapped I/O

Performance

```
template <typename coro_type>
struct test_coro
{
    static coro_type three()
    {
        co_await resume_background();
        co_await 0s;
    }

    static coro_type two()
    {
        co_await three();
    }

    static coro_type one()
    {
        co_await two();
    }
};
```

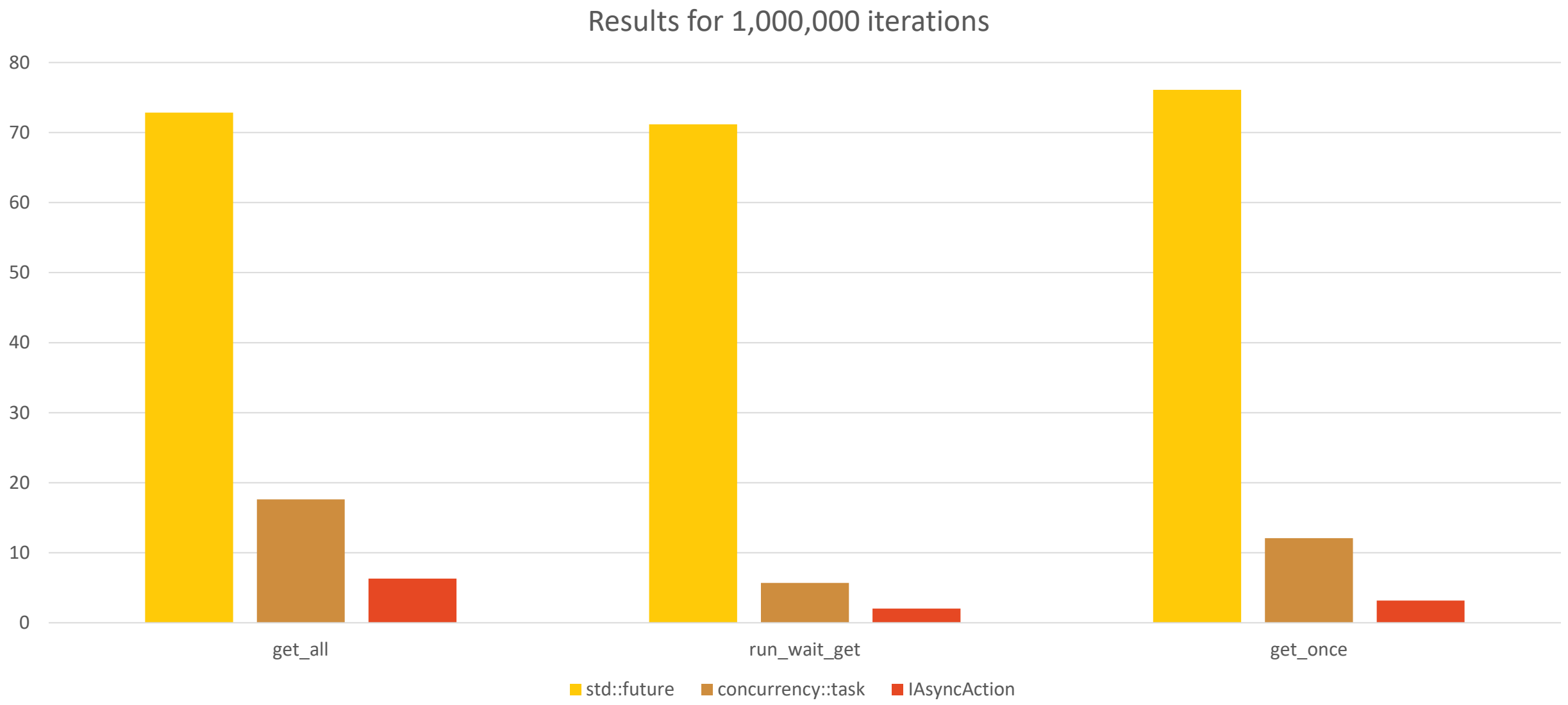
```
static void get_all()
{
    for (uint32_t i = 0; i != iterations; ++i)
    {
        one().get();
    }
}
```

```
template <typename coro_type>
struct test_coro
{
    static coro_type three()
    {
        co_await resume_background();
        co_await 0s;
    }

    static coro_type two()
    {
        co_await three();
    }

    static coro_type one()
    {
        co_await two();
    }
};
```

```
template <typename InIt>
static coro_type wait_all(
    InIt first,
    InIt last)
{
    for (; first != last; ++first)
    {
        co_await *first;
    }
}

static void run_wait_get()
{
    std::vector<coro_type> v(iterations);

    for (coro_type & coro : container)
    {
        coro = one();
    }
    wait_all(v.begin(), v.end()).get();
}
```

```
template <typename coro_type>
struct test_coro
{
    static coro_type three()
    {
        co_await resume_background();
        co_await 0s;
    }

    static coro_type two()
    {
        co_await three();
    }

    static coro_type one()
    {
        co_await two();
    }
};
```

```
static coro_type run_all()
{
    for (uint32_t i = 0; i != iterations; ++i)
    {
        co_await one();
    }
}

static void get_once()
{
    run_all().get();
}
```


Results for 1,000,000 iterations
80
70
60
50
40
30
20
10
0
get_all
run_wait_get
get_once
std::future
concurrency::task
IAsyncAction

More Information

Web: https://moderncpp.com

Email: kenny.kerr@microsoft.com

Twitter: @KennyKerr & @JamesMcNellis

\0