luncliff.github.io

Exploring MSVC Coroutine

> Written in 2017/02/17

Reference

Proposal

- N3858
- N3977
- N4134
- N4402

Visual C++ Team Blog

- Coroutines in Visual Studio 2015– Update 1
- More about resumable functions in C++

Video

- CppCon 2016 : Gor Nishanov “C++ Coroutines: Under the covers”
- CppCon 2016 : James McNellis “Introduction to C++ Coroutines”
- CppCon 2016 : Kenny Kerr & James McNellis “Putting Coroutines to Work with the Windows Runtime”
- CppCon 2016 : John Bandela “Channels - An alternative to callbacks and futures”
- CppCon 2015 : Gor Nishanov “C++ Coroutines - a negative overhead abstraction”
- Meeting C++ 2015 : James McNellis “An Introduction to C++ Coroutines”
- Meeting C++ 2015 : Grigory Demchenko “Asynchrony and Coroutines”
- CppCon 2014 : Gor Nishanov “await 2.0: Stackless Resumable Functions”

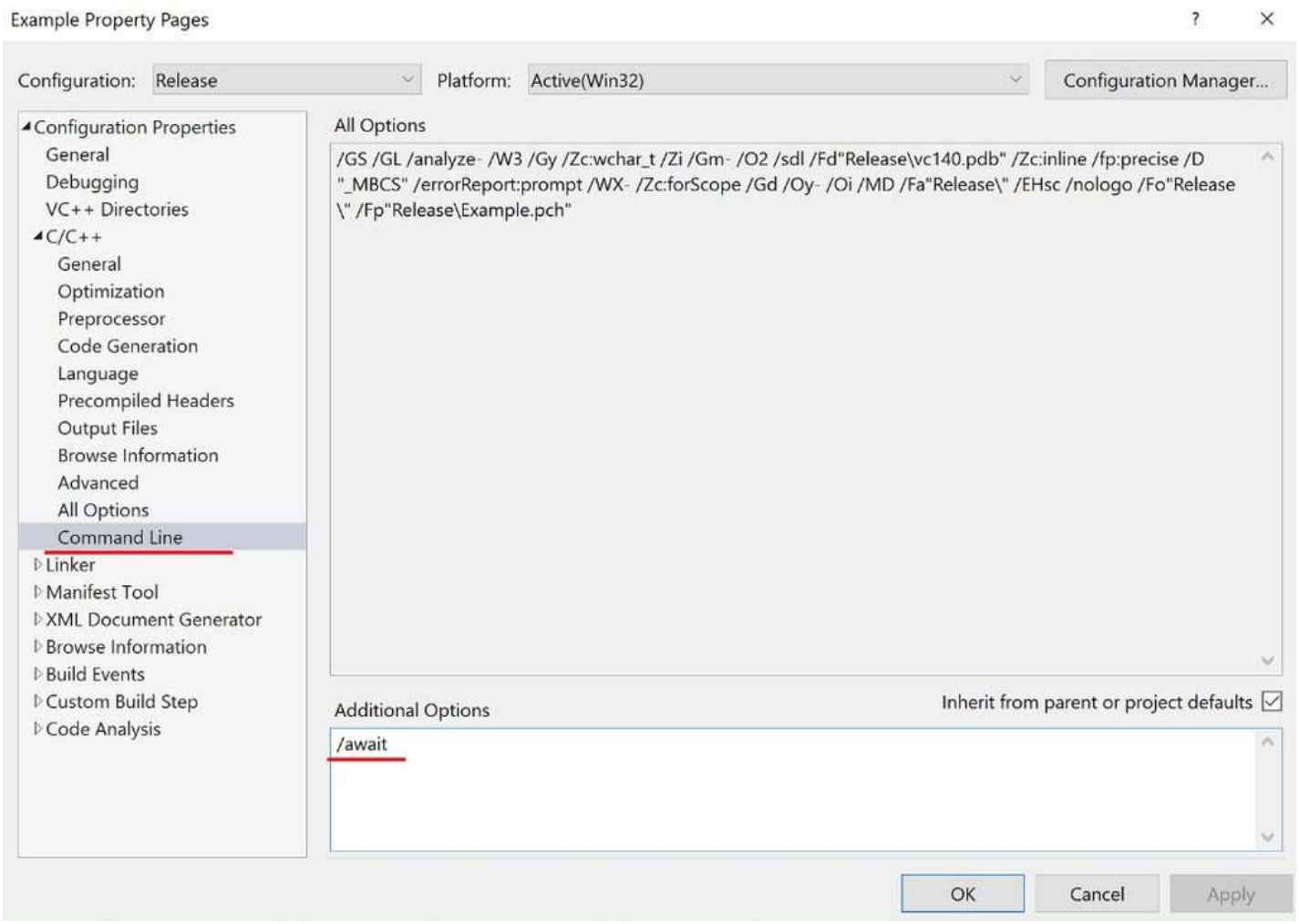
GitHub

- kirkshoop/**await**

Caution

To try this feature, you need 3 things.

- Visual Studio 2015 Up3 or later.
- `/await` Compiler option
- `<experimental/*>` header files to include.



Definition

So, what is coroutine?

Old Fashion : The Machine Level

For me, the story starts from **The Art of Computer Programming**, written by Donald Knuth.

Basically, the procedure call is `goto` . But, `goto` where? As you know, program can be thought as a sequence of states. And program's state, which is defined by programmer, is composition of machine environment and memory. It can be finite, or infinite.

- *Program State = Environment X Memory (Cartesian Product)*

Here, instructions are transition between states. And routine is a ordered group of instructions.

- *Instruction : Transition between states*
- *Routine : Ordered group of instructions*

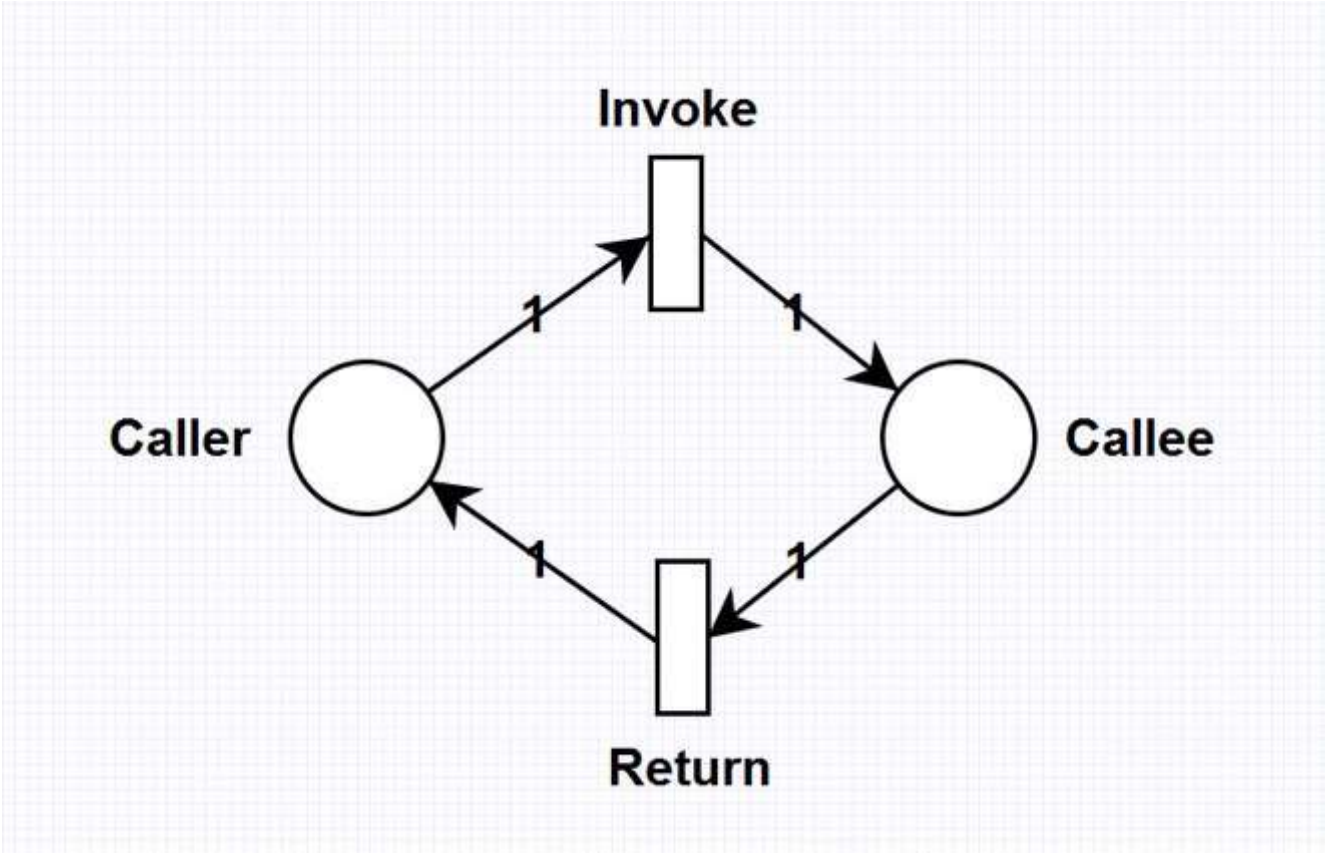
Now, let's go to subroutine & coroutine.

Concept : Relation

All programmers have their own mental model for codes. Lets start from the point. Mental model for subroutine and coroutine.

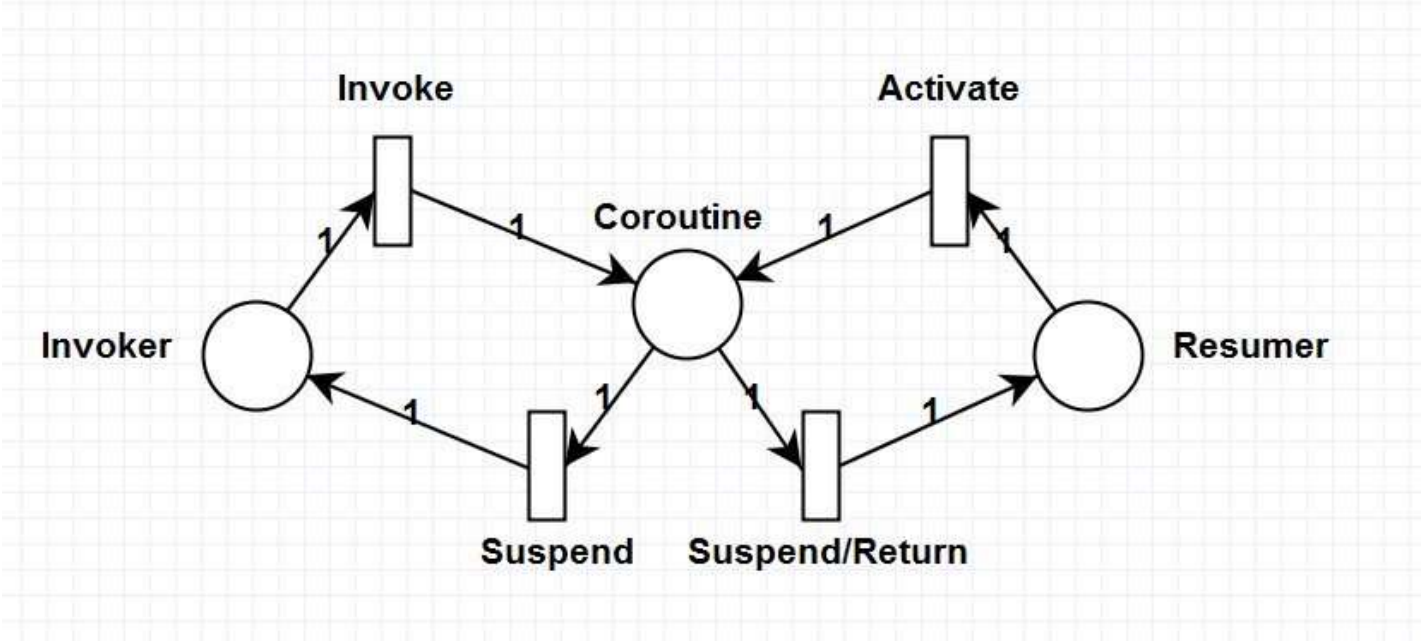
Subroutine : Caller-Callee

Caller expects some states (Pre-condition). And it knows which routine is required. Callee, with specified (in other words, well-defined) codes, do its work without consideration of its caller. It can be impure with side-effect. But it's about work, not the relation.



Coroutine : Activator-Activator

Coroutine is cooperative routine. In other worlds, it is aware of the other routines.



Notice that both are activated and activating by each other. Coroutine is activated by Invoker and Resuming routine. And by suspending or returning, coroutine activates its peer. In real, these **activations** are equal to jump(`goto` / `jmp`) instruction. So some suspension can be considered as invocation. Or there might not be significant difference between suspension and finalization.

If the point is entry(prologue) of routine, it is **invocation**(`call`).
If the flow goes to the middle of procedure, then it's **activation**(`resume`).

- *Invocation: Jump to start the routine*
- *Activation: Jump into a point of the routine*

Like subroutine, coroutines has specification for its pre-condition and post-condition. Therefore it must be designed with consideration of modification on program state. But after its work, it preserves the state and `goto` another point. This is **suspension**(`yield`). Of course, this can be return. In that case, it just finalizes and then `return` to its caller. (Since they have activation records)

- *Suspension: Jump to another point without finalization*
- *Finalization: Destroy(cleanup) all function resources*

Difference?

So, the major difference of coroutine/subroutine is that coroutine supports more operations. This is why Knuth summarized in his book.

> "Subroutines are special cases of ... coroutines" - Donald Knuth

Let's see that in table form.

Operation	Subroutine	Coroutine	
Invoke	o	o	General procedure start
Activate	x	o	`goto` a specific point of procedure
Suspend	x	o	Yield current control flow
Finalize	o	o	Cleanup and `return`

Clearly, subroutine is limited but easy and simple in relationship. However, coroutine have more operations, but might be complex because it suspension and activation should be managed.

- *Routine: Group of instructions that modify the program state to another with defined statements*
- *Subroutine: Routine that supports 2 operations*
    i. Invoke
    ii. Finalize
- *Coroutine: Routine that supports 4 operations*
    i. Invoke
    ii. Activate
    iii. Suspend
    iv. Finalize

Example

Knuth wrote how this is expressed in code. You don't have to understand all MIX instructions. Unlike theseday's subroutine, operations are usually `JMP` without abstraction.

What I want to explain is that there is a gap between High-level and Assembly languages. The key point is **coroutine has multiple points for entry and exit**. Also, they have some implicit states.

```
// MIX example, modified
// The Art of Computer Programming 1.4.2.
// For detail, read the book. :D

// Subroutine for character input
READER  EQU     16
INPUT   ORIG    *+16
NETCHAR STJ     9F
        JXNZ    3F          // ---> (3H)
1H      J6N     2F          // ---> (2H)
        IN      INPUT(READER)
        JBUS    *(READER)
        ENN6    16
2H      LDX     INPUT+16,6  // <--- (1H) J6N 2F
```

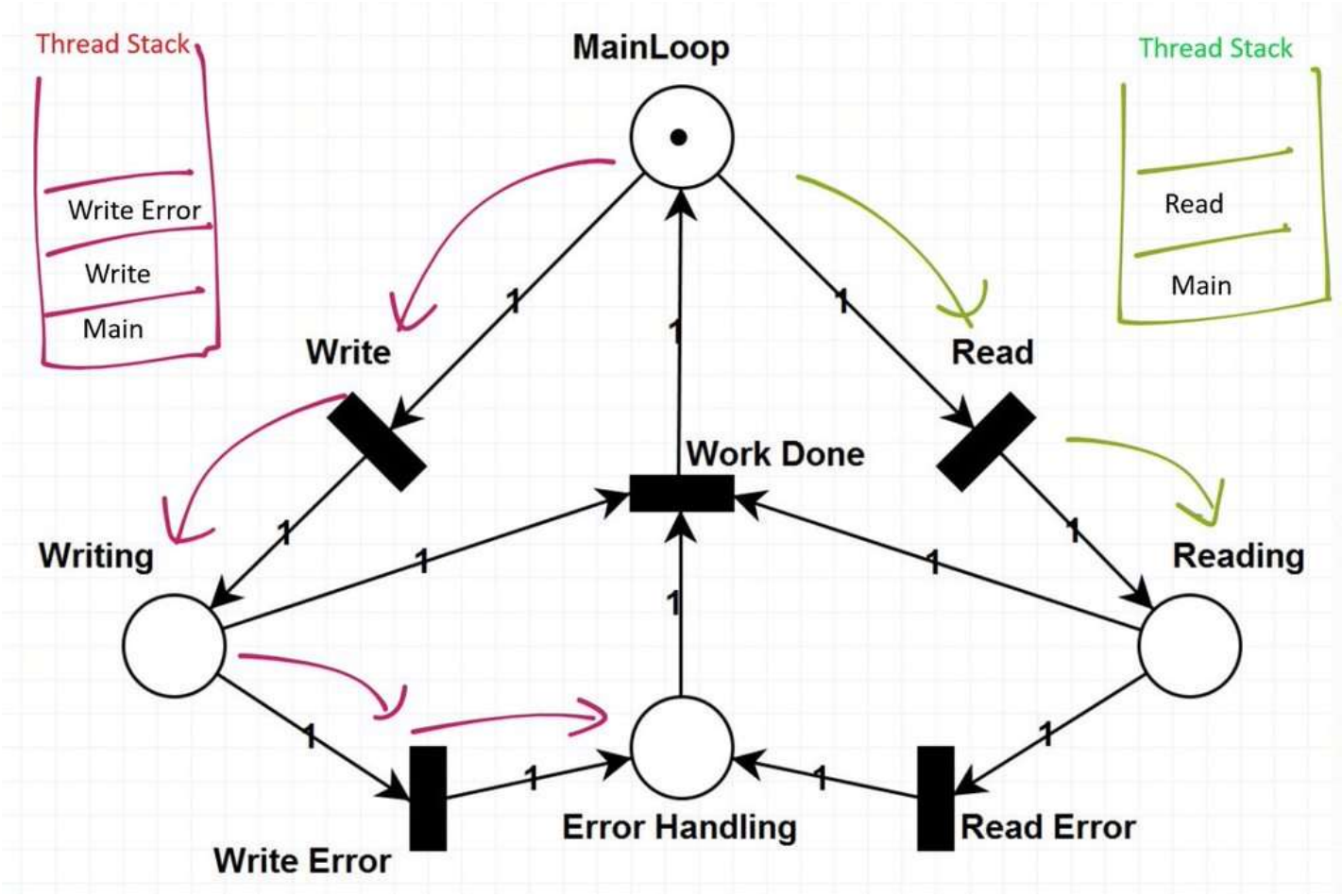
```
            INC6    1
    3H      ENTA    0           // <--- JXNZ 3F
            SLAX    1
    9H      JANZ    *
            JMP     NEXTCHAR+1

    // First coroutine
    2H      INCA    30          // <--- JGE 2B
            JMP     OUT         // ---> OUT1
    IN1     JMP     NEXTCHAR    // ---> NETCHAR
                                //  <--
            DECA    30
            JAN     2B          // ---> (2H)
            CMPA    =10=
            JGE     2B          // ---> (2H)
            STA     *,1(0:2)
            ENT5    *
            JMP     NEXTCHAR    // ---> NETCHAR
                                //  <--
            JMP     OUT         // ---> OUT1
                                // <--- J5NN *-2
            DEC5    1
            J5NN    *-2         // ---> JMP OUT
            JMP     IN1         // ---> (IN1)

    // Second coroutine
            ALF
    OUTPUT  ORIG    *+16
    PUNCH   EQU     17
    OUT1    ENT4    -16         // <--- JMP  OUT
            MOVE    -1,1(16)
    1H      JMP     IN          // ---> IN1
            STA     OUTPUT+16,4(1:1)
            CMPA    PERIOD
            JE      9F          // ---> (9H)
            JMP     IN          // ---> IN1
            STA     OUTPUT+16,4(2:2)
            CMPA    PERIOD
            JE      9F          // ---> (9H)
            JMP     IN          // ---> IN1
            STA     OUTPUT+16,4(2:2)
            CMPA    PERIOD
            JE      9F          // ---> (9H)
            INC4    1
            J4N     1B          // ---> (1H)
    9H      OUT     OUTPUT(PUNCH)
            JBUS    *(PUNCH)
            JNE     OUT1        // ---> OUT1
            HLT
    PERIOD  ALF     .
```

Motivation : Programming Model

It's a stack!

> "It's A Trap!" - Admiral Ackbar

These days, we are familiar with high-level languages with runtime stack. It's fancy, well-defined, and automated. Think of structured, stack-ful program's model. **In the model, the program is basically a mathematical expression tree** that flows in depth-first order. For instance, `main` function finished at some time point and then it returns a value. Like figure, functions in this model embeds a sequence of statements when it is written.



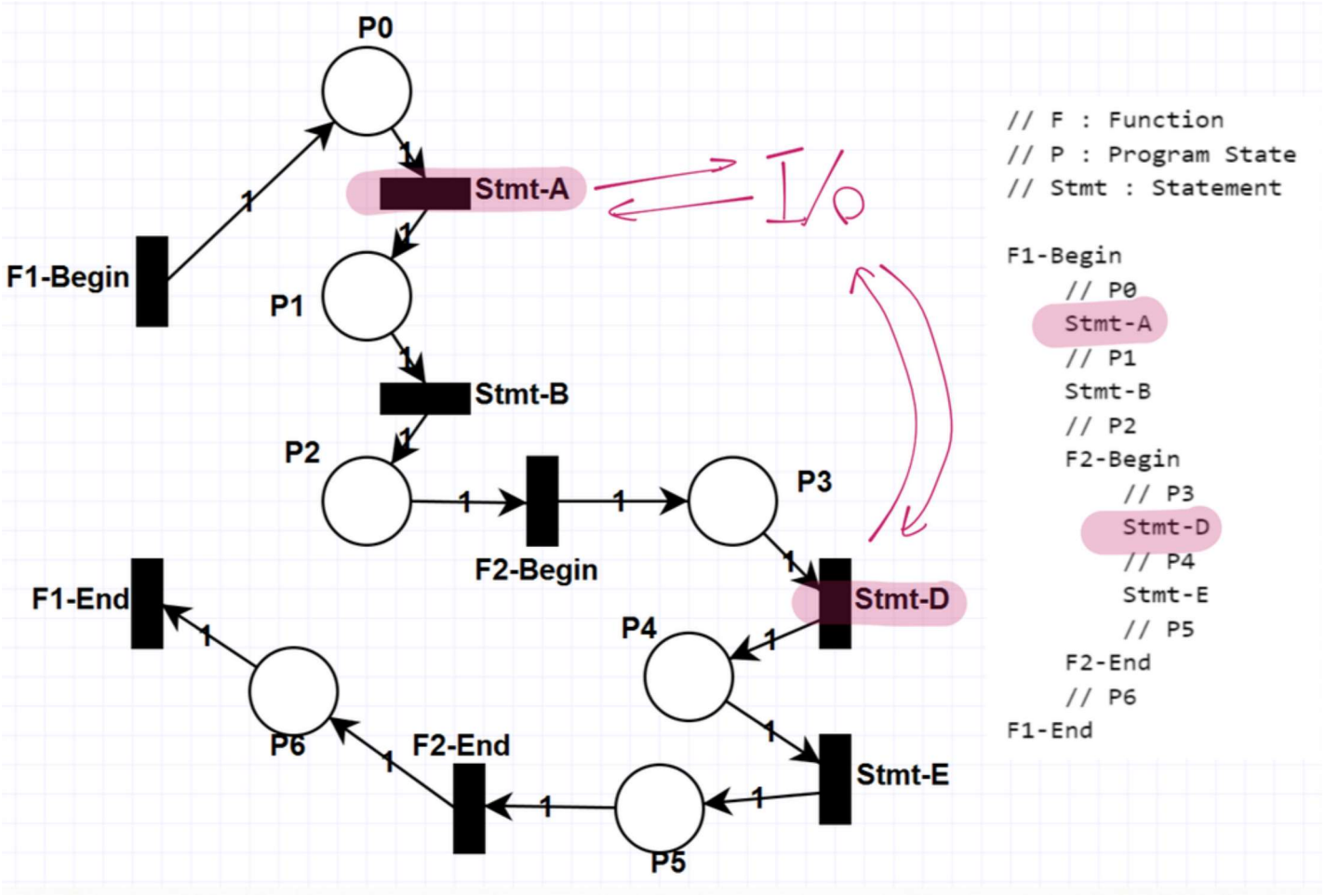
To reach the statement B, we have to finish the traversal for statement A. Then, the program state is affected by statement A. Memory locations will containing result values. Environments like registers will be set as defined.

Blocking

But the dependency became a problem.

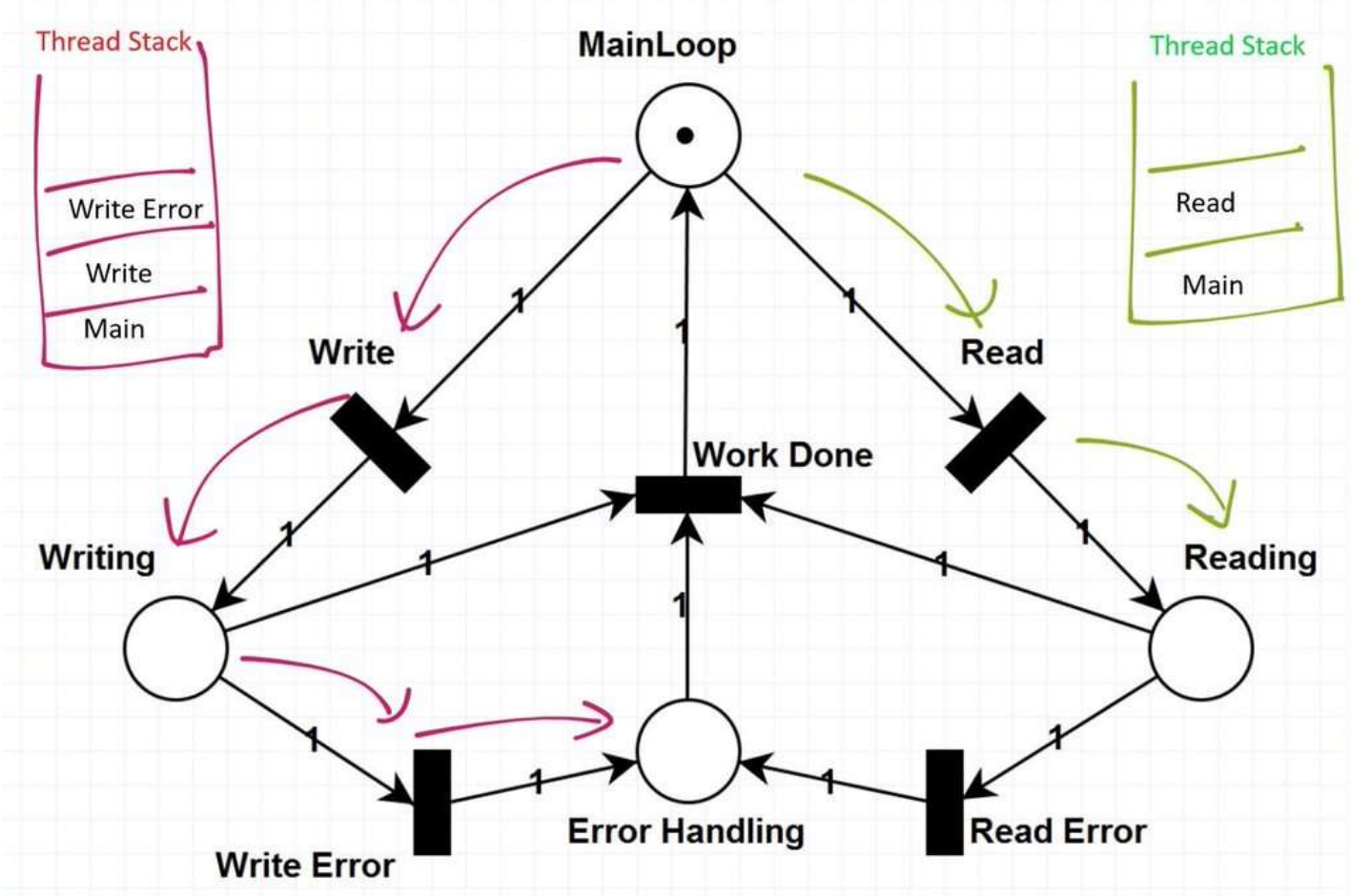
Remind that tree model follows mathematics. If we have all values in the expression, the calculation can be processed without blocking. Since we don't have infinite resource, we can't prepare all values when the flow starts.

So, we need some In/Out operations. which is extremely slow. Waiting its result blocks our control flow.



We don't want to stop our program. We want responsiveness. That's why we started to use non-blocking operations and to apply some asynchronous programming models. We generalized the program tree to a graph form. Which is another notation for state machine.

Now, **the program is graph of states**. And our processors traverse over it.



> Processor can be OS Process, OS Thread, or some language supported abstraction such as Goroutine and Erlang process. But, Let me skip them, we are thinking about coroutine now.

Persistence

The point is, we have to deliver some **context** between vertices of the graph. But subroutine can't do this. Because it always destroyes(finalizes) itself.

Subroutine is lack of persistence. It returns to caller. That's how runtime stack works. But it disabled us to pass contexts between states. Think of **callback hell** to enable that context forwarding.

> "James. This code has a bug. Can you fix it?" - Gor Nishanov, CppCon2015

```
std::future<int64_t> tcp_reader(int64_t total)
{
    struct reader_state {
        std::array<char, 4096> _buffer;
        int64_t                _total;
        tcp::connection        _connection;

        explicit reader_state(int64_t total) :
            _total(total) {}
    };
```

```
    auto state = std::make_shared<reader_state>(total);

    return tcp::connect("127.0.0.1", 1337).then(
        [state](std::future<tcp::connection> the_connection) {
            state->_connection = std::move(the_connection.get());
            return do_while([state]() -> std::future<bool> {
                if (state->_total <= 0) {
                    return std::make_ready_future(false);
                }
                return state->conn.read(state->_buffer.data(),
                                        sizeof(state->_buffer)).then(
                    [state](std::future<int64_t> bytes_read_future) {
                        int64_t bytes_read = bytes_read_future.get();
                        if (bytes_read == 0) {
                            return std::make_ready_future(false);
                        }
                        state->_total -= bytes_read;
                        return std::make_ready_future(true);
                    });
            });
        });
}
```

Approach

We need some persistent function. Reminding coroutine suspend/yield its control without finalization, the answer is crystal clear. Coroutine can take the role. But how can we migrate that assembly concept into our stack world?

Let's change the question.

1. What should be persistent?: **Context**
2. What is context?: **The routine's state**
3. What makes the state?: **Environment & Memory**
4. Where are they?: **Function Frame**

Henceforce, our migration starts from **how to make those frames persistent**. If you can't get what it is, visit this page.

Stackful? Stackless!

> "Why do you rob banks?" "That's where the money is." - Bank robber

Stackful or Segmented Stack

It differes upon language, but C++ places function frames in runtime stack. To preserve some frame, we have to dump it and store it in some location. Or we can make a smaller chunk of space and use it as a side(alternantive) stack for function.

But let's bypass them. Our focus, MSVC coroutine is stackless.

Stackless - Function Frame

N3858 wrote about resumable function's implementability. It suggested heap-allocated activation frames.

But frame is just a structure. Remind that we need 4 operations for coroutine. Those are declared as compiler intrinsic.

```
// Header File : <experimental/resumable>

// ...
// intrinsics used in implementation of coroutine_handle
extern "C" size_t _coro_resume(void *);
extern "C" void   _coro_destroy(void *);
extern "C" size_t _coro_done(void *);

#pragma intrinsic(_coro_resume)
#pragma intrinsic(_coro_destroy)
#pragma intrinsic(_coro_done)

// ...
// resumable functions support intrinsics
extern "C" size_t _coro_frame_size();
extern "C" void * _coro_frame_ptr();
extern "C" void   _coro_init_block();
extern "C" void * _coro_resume_addr();
extern "C" void   _coro_init_frame(void *);
extern "C" void   _coro_save(size_t);
extern "C" void   _coro_suspend(size_t);
extern "C" void   _coro_cancel();
extern "C" void   _coro_resume_block();

#pragma intrinsic(_coro_frame_size)
#pragma intrinsic(_coro_frame_ptr)
#pragma intrinsic(_coro_init_block)
#pragma intrinsic(_coro_resume_addr)
#pragma intrinsic(_coro_init_frame)
#pragma intrinsic(_coro_save)
#pragma intrinsic(_coro_suspend)
#pragma intrinsic(_coro_cancel)
#pragma intrinsic(_coro_resume_block)
// ...
```

So there is nothing we can touch.(Unless you are in MSVC team... right?) But if you want more detail, follow the references above and watch CppCon talks.

you may noticed there are 2 kinds of concept. Frame and Block.

- Coroutine
  i. `_coro_resume`
  ii. `_coro_destroy`
  iii. `_coro_done`
  iv. `_coro_save`
  v. `_coro_suspend`
  vi. `_coro_cancel`
  vii. `_coro_resume_addr`
- Frame
  i. `_coro_frame_ptr`
  ii. `_coro_frame_size`
  iii. `_coro_init_frame`
- Block
  i. `_coro_init_block`
  ii. `_coro_resume_block`

> It is not clear for me now. Let me update this section later. In my opinion, the frame is activation record for the function (In the case, resumable function). And block is space for function code. Therefore, it will contain local/captured arguments, And it's size will depends on maximum stack size of the function. ... Probably.

The C++ Coroutine : Resumable Function

At this point, we have to define 2 terms.

- *Coroutine : **Concept**. Routine that supports 4 operations*
- *Resumable Function : **Implementation** of coroutine concept in C++ world*

And the following table is about how to use its operations

Operation	Subroutine	Coroutine
Invoke	`func(args)`	`func(args)`
Activate	x	`resume()`

Operation	Subroutine	Coroutine
Suspend	x	`co_yield` / `co_await`
Finalize	`return`	`co_return`

Invoke

For function call, there is no change. But the resumable function returns `std::future<T>`.

Finalize

Cleanup and return point can be specified the well-known keyword, `return`. For same purpose, in resumable function, `co_return` statement is used.

Suspend

When you wan to suspend funtion and yield its control, `co_yield` expression and `co_await` operator can do that. By adding 1 of them, the suspend/resume point is specified and MSVC will treat the function as coroutine. (Also, `co_return` statement can be used)

Activate

When we have to activate suspended function, we will use `coroutine_handle` and its memeber function, `.resume()`. This is an interface to compiler intrinsic `_coro_resume` above.

Code

> Example codes are inspired by kirkshoop's repository.

In short, if we follows MSVC's requirement, we can use C++ coroutine. But notice that the title of proposal is **resumable**(therefore, persistent) function. However, its category differs to which meaning you want to carry with. Before that, let's start with compiler's requirement.

Promise : Compiler's Requirement

You may already know about `std::future<T>` and `std::promise<T>`, and relation of the pair. Usually, `future<T>` is something like "I will return value or exception **later**" (See `std::async()`). And the value/exception is transferred via `promise<T>`. It is, "I kept the **promise** for the value".

And I already explained that resumable returns `future<T>`. Literally, This is not weird because coroutine can suspends itself and therefore its activator can't sure that it has returned. However,

with the signature, programmer can still explicitly specify for return type just like conventional subroutine.

But here, the promise is for resumable function. See N4402. You may ask why this is necessary. The answer is that resumable function is **abstraction**.

Because of stack, we had migrated the coroutine concept but not assembly-like flat code. Our resumable function lives in heap and to pass value from heap space(from function frame) to stack space(to `future` or awaitable), there must be some **interface**. For both compiler and programmer.

Resumable Promise Requirement

I will explain how this interface is used soon. According to N4402, the promise type for resumable function should support operations in the table below. I summarized note column. For detail, please read the document.

Expression	Note
`P{}`	Promise must be default constructible
`p.get_return_object()`	The return value of funtion. It can be `future<T>` , or some user-defined type.
`p.return_value(v)`	`co_return` statement. Pass the value `v` and the value will be consumed later.
`p.return_value()`	`co_return` statement. Pass `void` . Can be invoked when the coroutine returns. And calling this can be thought as “No more value”.
`p.set_exception(e)`	Pass the exception. It will throw when the resumer activates the function with this context.
`p.yield_value(v)`	`co_yield` expression. Similar to `return_value(v)` .
`p.initial_suspend()`	If return `true` , suspends at initial suspend point.
`p.final_suspend()`	If return `true` , suspends at final suspend point.

Some operations are related to `coroutine_traits<T>` . Later section will describe its flexibility. It is pretty abstract for now, but we are in progress. Lets move our focus to usecases of C++ coroutine.

Generator

Concept

- *Generator = Iterator + Resumable Function*

See? There is an iterator. `generator<T>` is abstraction for sequence of values, which are generated by resumable function. The function is persistent, and contains some unique context in its frame.

Example

The following code is simple generator for fibonacci numbers.

```
#include <experimental/generator>

// Make a generator for `N` fibonacci numbers
auto fibonacci(int n) noexcept
    -> std::experimental::generator<int>
{
    int f1 = 0;
    int f2 = 1;

    for (int i = 0; i < n; ++i) {
        // Set the value and suspend
        co_yield f1;

        // Calculate next fibo and shift
        int f3 = f1 + f2;
        f1 = f2;
        f2 = f3;
    }
    co_return;  // No more value
}

void usecase()
{
    // A sequence of 10 fibo numbers
    for (int fibo : fibonacci(10))
    {
        // 0, 1, 1, 2, 3, 5, 8 ... 34
    }
}
```

How this can be possible? Well, lets see the definition of `generator<T>` .

Detail

Here is a skeleton of `generator<T>` . We can see that it supports `iterator` and `promise_type` . and `begin()` / `end()` function to support Range-based for loop

```
template <typename _Ty, typename _Alloc = allocator<char>>
struct generator
{
    // Resumable Promise Requirement
    struct promise_type;
private:
    // Handle for Resumable Funtion
    coroutine_handle<promise_type> _Coro = nullptr;
public:
    struct promise_type { /*...*/ };
    struct iterator { /*...*/ };

    iterator begin()
    {
        // If we have handle, we can resume it to get the value.
        if (_Coro) {
            _Coro.resume();
            // The function returned?
            if (_Coro.done())
                return {nullptr}; // return end();
        }
        // The function is not finished.
        // Will be reused later...
        return {_Coro};
    }

    iterator end()
    {
        // No handle, No more value.
        return {nullptr};
    }

    ~generator()
    {
        // RAII : Destroy function frame
        if (_Coro) {
            _Coro.destroy();
        }
    }

};
```

So, the actual usecase function will be like this.

```
void usecase()
{
    // A generator for 10 fibo numbers
    generator<int> gen = fibonacci(10);
```

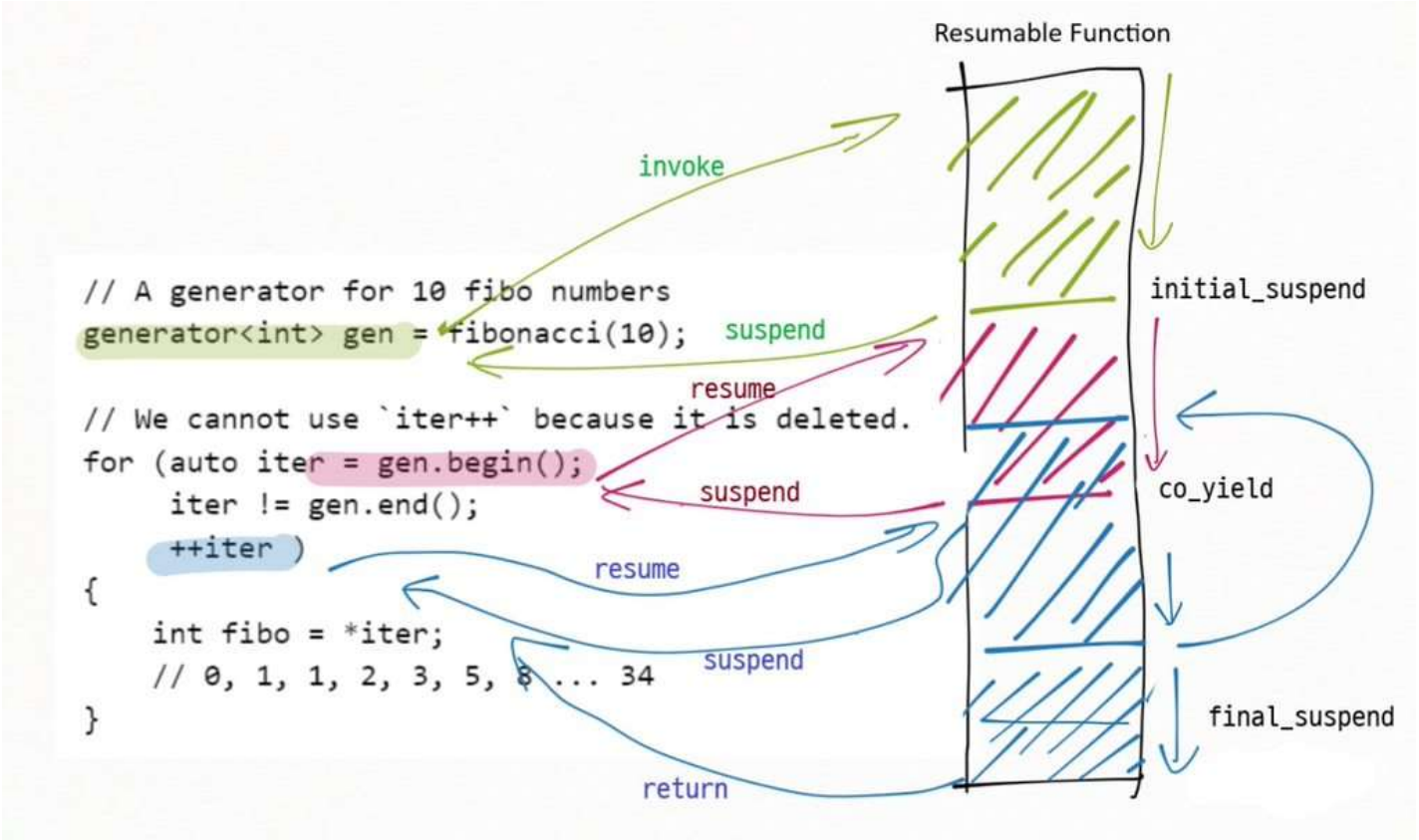
```
    // We cannot use `iter++` because it is deleted.
    for (auto iter = gen.begin();
         iter != gen.end();
         ++iter )
    {
        int fibo = *iter;
        // 0, 1, 1, 2, 3, 5, 8 ... 34
    }
}
```

Pretty simple with the iterator! Let's hack the iterator then...

```
template <typename _Ty, typename _Alloc = allocator<char>>
struct generator
{
    // Iterator interface
    struct iterator : _STD iterator<input_iterator_tag, _Ty>
    {
        // Resumable function handle
        coroutine_handle<promise_type> _Coro;

        iterator(nullptr_t);
        iterator(coroutine_handle<promise_type> _CoroArg);

        // When we move next...
        iterator &operator++()
        {
            // Make the function yield next value
            _Coro.resume();

            // If done, just release.
            // Generator will destroy it later...
            if (_Coro.done())
                _Coro = nullptr;
            return *this;
        }

        // This is MSVC comment....
        // In short, post increment can overlap the handle in iterator.
        // And it can leak the last value in coroutine frame.
        // ---- ---- ---- ---- ----
        // generator iterator current_value
        // is a reference to a temporary on the coroutine frame
        // implementing post increment will require storing a copy
        // of the value in the iterator.
        iterator operator++(int) = delete;
        //{
        //      auto _Result = *this;
        //      ++(*this);
```

```
        //      return _Result;
        //}

        // We use `promise` to get the value.
        // It is pointing the value in function frame
        _Ty const &operator*() const
        {
            return *_Coro.promise()._CurrentValue;
        }

        _Ty const *operator->() const;

        bool operator==(iterator const &_Right) const;
        bool operator!=(iterator const &_Right) const;

    };
};
```

At this point, we can understand that iterator is resuming the function repetitively. And acquiring value from `generator<T>`'s `promise_type`. So in this case, `usecase` function is **both invoker and resumer for the generator**.



Compiler's View

Now, the keystone `promise_type` is on the stage. It has more codes but I will skip them for simplicity.

```
template <typename _Ty, typename _Alloc = allocator<char>>
struct generator
{
    // Resumable Promise Requirement for `generator`
    struct promise_type
    {
        // it knows where the value is... in frame.
        _Ty const *_CurrentValue;

        // It returns `promise_type`.
        // The reason will be explained in next section.
        promise_type &get_return_object()
        {
            return *this;
        }

        // If return `true`, suspends at initial suspend point.
        // So we suspends at *some* point
        bool initial_suspend()
        {
            return (true);
        }

        // If return `true`, suspends at final suspend point.
        // So we suspends at *some* point
        bool final_suspend()
        {
            return (true);
        }

        // Support `co_yield` expression
        void yield_value(_Ty const &_Value)
        {
            // point the value
            _CurrentValue = _STD addressof(_Value);
        }
        // ...
    }

    // ...
}
```

The `promise_type` supports 4 functions in requirement. `get_return_object`, `initial_suspend`, `final_suspend`, and `yield_value`.

In short, `co_yield` is equal to "Set the value and suspend". Compiler will change the expression like following.

- `co_yield` : `co_await` + `p.yield_value(x)`

Therefore, `co_yield` with empty expression is **impossible**. Because `x` can't be `void`. Since generator is abstraction of sequence, this is natural constraint.

```
#include <experimental/generator>

auto fibonacci(int n) noexcept
    -> std::experimental::generator<int>
{
    // ...
    for (int i = 0; i < n; ++i) {

        // Compiler changes the expression to...
        // co_yield f1;
        p.yield_value(f1);  // Set the value pointer
        co_await suspend_always{}; // And suspend.

        // ...
    }
    co_return;  // No more value
}
```

Notice that there is no `promise_type p{}` or something else. Its secret will be covered in next section.

Resumable Function

This section starts from detail. The example code will be seen after explanation of awaitable concept.

Concept

- *Resumable Function : Heap Allocated Frame + Operation*
_Operation : `call`	`co_await`	`co_return`	`resume()` _

`generator<T>` was simple example. And you may remember that it contains `coroutine_handle`. Let's cover the type. `coroutine_handle` was `resumable_handle`.

Remembering that resumable function is C++ implementation of coroutine concept, it is alias of resumable function handle. If you have an experience of Windows API, you probably know what *handle* means. It is `void *`.

We already talked about how stackless coroutine can be implemented. MSVC uses heap allocated function frame and supports several intrinsics. **With the pointer to frame in heap, `coroutine_handle` adds some operation over it.**

Resumable Frame

Then, how does the frame look like? As I commented above, the frame of resumable function. It has head and body.

```
// TEMPLATE CLASS coroutine_handle
template <typename _PromiseT = void>
struct coroutine_handle;

// TEMPLATE CLASS coroutine_handle<void> - no promise access
template <>
struct coroutine_handle<void>
{
    // ---- ---- ---- ---- ----
    // Head of function frame
    //  - Fn    : Instruction address for resume operation
    //  - cdecl : Caller manages the frame. Callee won't touch it.
    //  - Index : Index to resumption point.
    //            0 is special value for `done()`
    //  - Flag  : ???
    struct _Resumable_frame_prefix
    {
        typedef void(__cdecl *_Resume_fn)(void *);
        _Resume_fn _Fn;
        uint16_t _Index;
        uint16_t _Flags;
    };
    // ---- ---- ---- ---- ----
    // Coroutine Frame will include...
    //  - Promise
    //  - Captured arguments
    //  - Function body
    //      - Local Variables and Temporaries
    //  - Platform context
    //      - Registers
    // ---- ---- ---- ---- ----

protected:
    _Resumable_frame_prefix *_Ptr;

public:
    coroutine_handle() noexcept = default;
    coroutine_handle(std::nullptr_t) noexcept;
    coroutine_handle &operator=(nullptr_t) noexcept;

    // Import : the pointer to resumable handle
    static coroutine_handle from_address(void *_Addr) noexcept
    {
        coroutine_handle _Result;
        _Result._Ptr = reinterpret_cast<_Resumable_frame_prefix *>(_Addr);
```

```
        return _Result;
    }

    // Export : return the frame address
    void *address() const noexcept
    {
        return _Ptr;
    }

    explicit operator bool() const noexcept;
    void resume() const;
    void destroy();

    bool done() const
    {
        // REVISIT: should return _coro_done() == 0; when intrinsic is
        // hooked up
        return (_Ptr->_Index == 0);
    }

    // ...
};
```

Head : Like its name, `_Resumable_frame_prefix` is head of coroutine frame. MSVC uses fixed size of `sizeof(void *) * 2` (16 bytes in x64) for this struct.

- `Fn` : Note that this is declared as `cdecl` convention. Since `cdecl` specifies stack cleanup (in the case, frame destruction) is up to caller, the call of `Fn` won't destroy the frame.
- `Index` : Resumable function can have multiple resumption point. This is a index for the point.
- `Flag` : ??? Who are you ???

Body : N4402 describes what frame includes. Based on James McNellis's PPT, the compiler-generated frame body will be like the following code.

- Promise
- Captured arguments
- Function body
  - Local Variables and Temporaries
- Platform context
  - Registers

```
// Possible frame for fibonacci function
struct __frame_fibonacci
{
    // `_Resumable_frame_prefix`
```

```
    Frame_Prefix  _prefix;

    // Resumable Promise Requirement
    //   fibonacci returns `generator<int>`
    generator<int>::promise_type  _p;

    // Captured arguments
    int   _n; // fibonacci(n);

    int   _f1, _f2;     // Local variable
    int   _i,  _f3;     // Temporaries

    // Platform dependent storage
    //  for registers, etc.
};
```

Coroutine Traits

We didn't cover `p.get_return_object()` yet. This is the section for the topic. Let me explain `coroutine_traits` first...

```
// <experimental/resumable>

// TEMPLATE CLASS coroutine_traits
template <typename _Ret, typename... _Ts>
struct coroutine_traits
{
    using promise_type = typename _Ret::promise_type;
};
```

It's pretty simple. `coroutine_traits` requires `promise_type` to be implemented at compile time. And it must follow **Resumable Promise Requirement**.

I explaind that resumable function returns `future<T>`, but that was not correct. If there is a type that fulfills the `coroutine_traits`, it can be return type of Resumable function.

Let't go to **template specialization** of `coroutine_traits` for `std::future<T>`. Previous example, `generator<T>` implemented promise type in itself, but with the specialization trick, we can reuse some old types or try some custom type.

```
// <future>
#ifdef _RESUMABLE_FUNCTIONS_SUPPORTED

namespace std::experimental
{
```

```
    // Template Specialization for `std::future<T>`
    template<class _Ty, class... _ArgTypes>
    struct coroutine_traits<future<_Ty>, _ArgTypes...>
    {
        // defines resumable traits for functions returning future<_Ty>
        struct promise_type
        {
            // `std::promise<T>`
            promise<_Ty> _MyPromise;

            // Resumable function can return `std::future<T>`...
            future<_Ty> get_return_object()
            {
                return (_MyPromise.get_future());
            }

            // if `false`, we skip the initial suspend point.
            bool initial_suspend() const
            {
                return (false);
            }

            // if `false`, we skip the final suspend point.
            bool final_suspend() const
            {
                return (false);
            }

            // Pass the value through `std::promise<T>`
            template<class _Ut>
            void return_value(_Ut&& _Value)
            {
                _MyPromise.set_value(_STD forward<_Ut>(_Value));
            }

            // Pass the exception through `std::promise<T>`
            void set_exception(exception_ptr _Exc)
            {
                _MyPromise.set_exception(_STD move(_Exc));
            }
        };// struct promise_type
    };// coroutine_traits<T, Args...>

    // ...
    }// namespace std::experimental

    #endif
```

When MSVC meets `co_await` / `co_yield` / `co_return` , it applies `coroutine_traits` to enable compile-time check. Let's see the fibonacci example again. It will show what MSVC does.

> The code might be wrong. I will update it as I find correct mechanism.

```
#include <experimental/generator>

using namespace std;
using namespace std::experimental;

// Make a generator for `N` fibonacci numbers
auto fibonacci(int n) noexcept
    -> std::experimental::generator<int>
{
    using return_type = generator<int>;
    // At this point, compiler will generate code
    //  and check `return_type` fulfills promise requirement.
    using traits = coroutine_traite<return_type>;

    // Requirement : OK.
    // We can generate unique frame type for this function.
    // Possible frame for fibonacci function
    struct __frame
    {
        // `_Resumable_frame_prefix`
        Frame_Prefix  _prefix;

        // Resumable Promise Requirement
        //   fibonacci returns `generator<int>`
        generator<int>::promise_type  _promise;

        // Captured arguments
        int   _n; // fibonacci(n);

        int   _f1, _f2;     // Local variable
        int   _i,  _f3;     // Temporaries

        // Platform dependent storage
        //  for registers, etc.
    };

    // We are forwarding arguments to frame!
    // Let's call it context(ctx)
    __frame* ctx = new __frame{std::move(n)};
    // Generate return objet
    // In this case, `generator<int>`
    return_type __return = ctx->_promise.get_return_object();

    // if true, suspend.
    // if false, keep move...
    if( ctx->_promise.initial_suspend() ){ // always true
        // suspend...
    __initial_suspend_point:
```

```
    }

    // User code : use variables in frame(ctx)...
    // ---- ---- ---- ---- ----
    {
        ctx->_f1 = 0;
        ctx->_f2 = 1;

        for (ctx->_i = 0;
             ctx->_i < ctx->_n;
             ctx->_i++)
        {
            // Pass value through promise
            // co_yield f1;
            ctx->_promise.yield_value(ctx->_f1);

            // instructions for suspension with `ctx->_prefix`...
            // co_await suspend_always{};
        __suspend_resume_point_1:

            // Calculate next fibo and shift
            ctx->_f3 = ctx->_f1 + ctx->_f2;
            ctx->_f1 = ctx->_f2;
            ctx->_f2 = ctx->_f3;
        }

        // co_return;
        ctx->_promise.set_result();
        goto __final_suspend_point;
    }
    // ---- ---- ---- ---- ----

    if( ctx->_promise.final_suspend() ){ // always true
        // suspend...
    __final_suspend_point:
    }

    // Instructions for clean up...
}
```

So we could find out that compiler do **a lot of job** instead of us. And using frame pointer `ctx` is not that different from `this` pointer for memeber functions. Therefore, the cost for frame-based code will be negligible.

- `co_yield` : `co_await` + `p.yield_value(x)`
- `co_return` : `p.set_result(x)` + `goto final_suspend;`

Section Summary

We covered `co_return` and `co_yield` with `generator<T>` example. Generator was a combination of iterator and resumable function.

Resumable functions use `coroutine_handle` , which is basically raw pointer to function frame and some operations that rely on compiler intrinsics. The mechanism for frame-based approach was pretty similar to `this` pointer approach.

For compile-time(static) assertion, `coroutine_traits<T>` is used. It enforce the return type to support `promise_type` that fulfills resumable promise requirement. And there was a specialization especially for `std::future<T>` .

The Awaitable Concept

- *`operator co_await` = Syntactic Sugar + Resumable Function*

The last core of resumable function is `co_await` . It is unary operator.

In this semantics, **await is synonym of suspend**. And because of suspension, its return value becomes **asynchronous**.

This is the important point for our mental model. You may thought `async` & `await` of C# language, The difference in C++ world is that we can manage them **manually**, with our code and compiler support.

Example

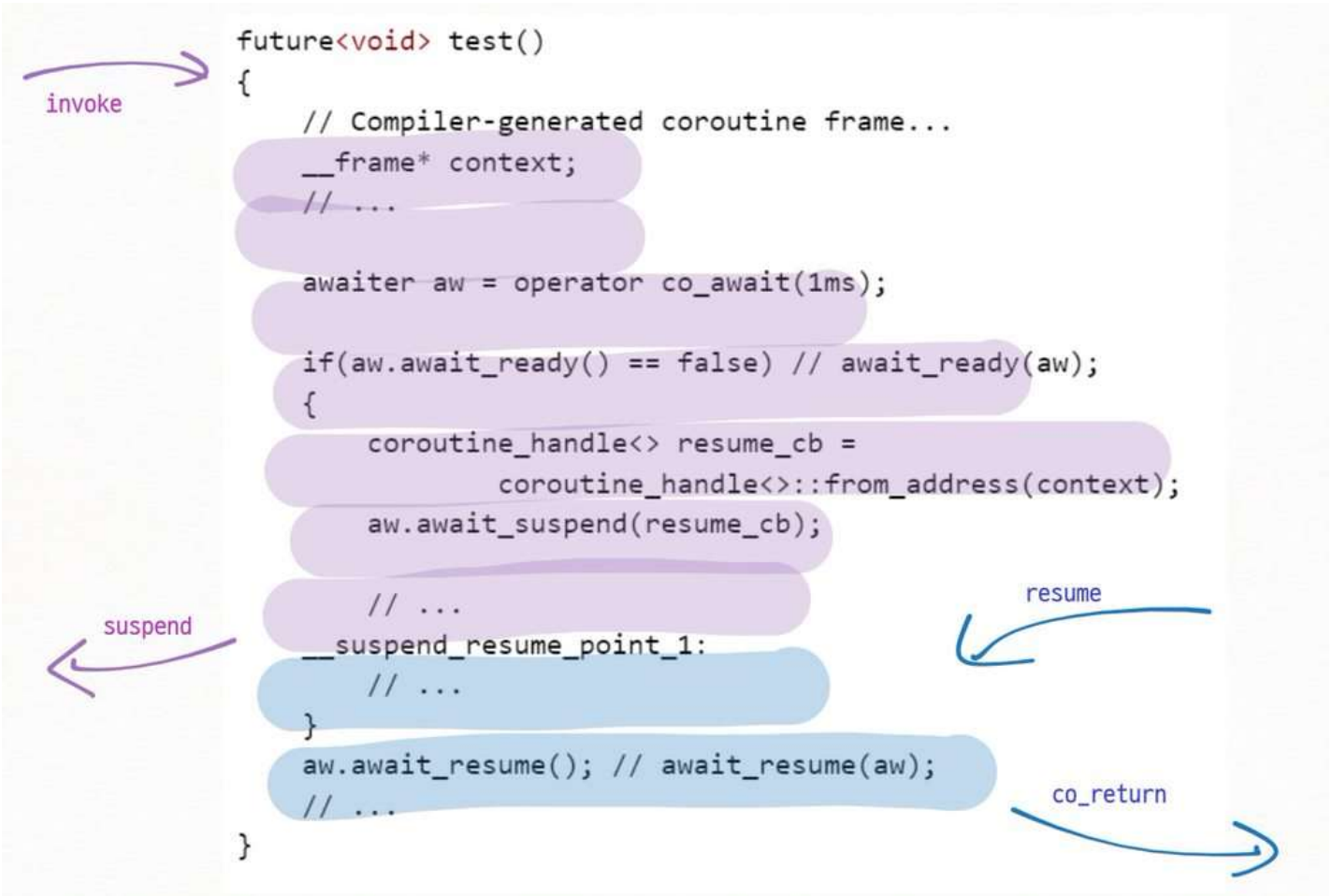
The code is sample from MSVC blog.

```
#include <iostream>
#include <chrono>
#include <future>
#include <windows.h>    // Windows Threadpool API

using namespace std;
using namespace std::literals;

// operator overload.
// co_await can't use primitive type parameter.
auto operator co_await(chrono::system_clock::duration duration)
{
    using namespace std::experimental;

    // Awaitable must implements 3 function.
    //  - bool await_ready();
    //  - auto await_suspend();
    //  - T    await_resume();
```

```
        class awaiter
        {
            static
            void CALLBACK TimerCallback(PTP_CALLBACK_INSTANCE,
                                        void *Context,
                                        PTP_TIMER)
            {
                // Callback Thread will resume the function
                coroutine_handle<>::from_address(Context).resume();
            }
            PTP_TIMER timer = nullptr;
            chrono::system_clock::duration duration;
        public:

            explicit
            awaiter(chrono::system_clock::duration d) : duration(d)
            {}
            ~awaiter() {
                if (timer) CloseThreadpoolTimer(timer);
            }

            // If not ready (`false`), invoke `await_suspend`
            // If ready (`true`), go to `await_resume` directly.
            bool await_ready() const
            {
                return duration.count() <= 0;
            }

            // Return might be ignored.
            bool await_suspend(coroutine_handle<> resume_cb)
            {
                int64_t relative_count = -duration.count();
                timer = CreateThreadpoolTimer(TimerCallback,
                                              resume_cb.address(),
                                              nullptr);
                // Set the timer and then suspend...
                SetThreadpoolTimer(timer, (PFILETIME)&relative_count, 0, 0);
                return timer != 0;
            }

            // Return T type's value after resumed.
            // T can be `void`.
            void await_resume() {}

        };
        return awaiter{ duration };
    }

    // Resumable Function
    future<void> test()
    {
```

```
    cout << this_thread::get_id() << ": sleeping…\n";

    // await for 1 millisecond...
    co_await 1ms;

    cout << this_thread::get_id() << ": woke up\n";
}

// This is normal subroutine
void usecase()
{
    test().get();
    cout << this_thread::get_id() << ": back in main\n";
}
```

It's flow is like the figure.



Awaitable Interface

`co_await` requires some interface for its operation. As we can see in code above, Awaitable interface should implement at least 3 functions.

- `await_ready`

- `await_suspend`
- `await_resume`

It can be both member and normal function. `future<T>` also implements this interface. so `future` can be an argument for `co_await`. But just like the code `await_suspend`, it can be inefficient. For user-defined type, simple overload is enough.

```
// <future>
namespace std
{

template<class _Ty>
bool await_ready(future<_Ty>& _Fut)
{
    return (_Fut._Is_ready());
}

template<class _Ty>
void await_suspend(future<_Ty>& _Fut,
                   experimental::coroutine_handle<> _ResumeCb)
{
    // change to .then when future gets .then
    thread _WaitingThread( [&_Fut, _ResumeCb]{
            _Fut.wait();
            _ResumeCb();
        });
    _WaitingThread.detach();
}

template<class _Ty>
auto await_resume(future<_Ty>& _Fut)
{
    return (_Fut.get());
}

}// namespace std
```

How about compiler's view? How does it change our code?

Compiler's View

- *`operator co_await` = Syntactic Sugar + Resumable Function*

I explained that `co_await` is syntatic sugar. If operator's argument type implements awaitable concept appropriately, the code will be modified by compiler. Revisiting previous example, `test` function which awaits for 1 millisecond, let's see how it works...

```
// Before...
// ---- ---- ---- ---- ----
future<void> test()
{
    // ...
    co_await 1ms;
    // ...
}

// After...
// ---- ---- ---- ---- ----
// With `co_await` keyword in function body,
// compiler will make this function to resumable.
future<void> test()
{
    // Compiler-generated coroutine frame...
    __frame* context;
    // ...

    // Awaitable type implements awaitable interface.
    // Precisely, this will be temporary variable in frame.
    awaiter aw = operator co_await(1ms);

    // Do we have expected value now?
    // If not, suspend.
    // If ready, skip the suspend and resume directly.
    if(aw.await_ready() == false) // await_ready(aw);
    {
        // Wrap the function frame to `coroutine_handle` type
        coroutine_handle<> resume_cb =
                coroutine_handle<>::from_address(context);

        aw.await_suspend(resume_cb);

        // some instructions....
    __suspend_resume_point_1:
        // Since resume point is in this scope,
        // there won't be suspension if ready()==true.
    }

    // If await_resume has return type...
    // auto value = aw.await_resume();
    aw.await_resume(); // await_resume(aw);

    // ...
}
```

So there are some points for each of interface functions.

- `await_ready` : *"Do we have return value now?"*

`await_ready` makes the branch. It asks value for `await_resume` is ready. If not, the control flows into the scope of `if` statement. If it's ready, `await_ready` returns `true` and `await_resume` will be invoked directly. Compiler can optimize the code if it returns `false` always. In the case, we don't need suspension and therefore frame wrapping becomes unnecessary. The coroutine will become normal subroutine after optimization steps.

In suspension scope, we must ready for suspension. Since `coroutine_handle<>` is just a raw pointer to frame, we can wrap it easily with static function `coroutine_handle<>::from_address` .

- `await_suspend` : *"The routine will suspend soon. Do what you have to do!"*

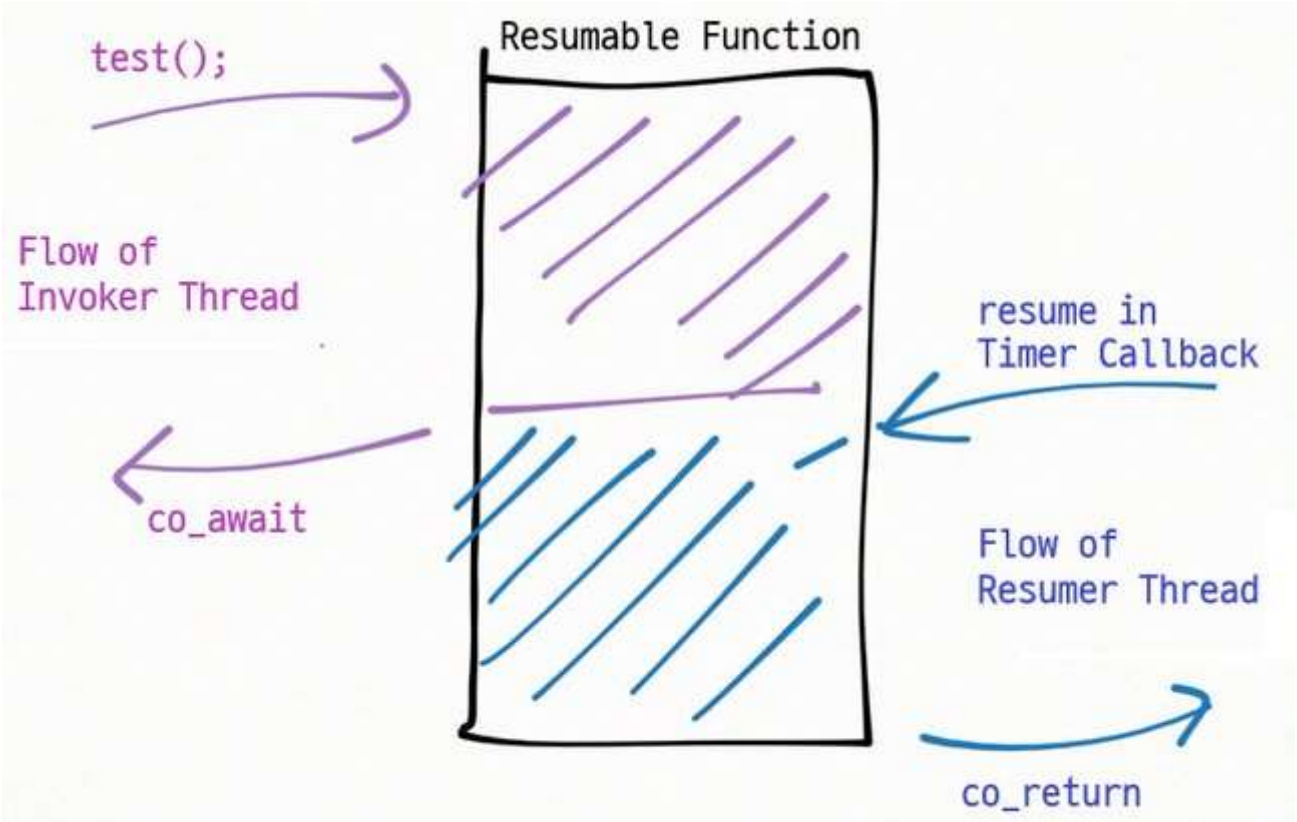
`await_suspend` receives `coroutine_handle<>` for its argument. With this function, we can interleave our code for suspension handling.

```
// The example code set the Windows Thread Pool Timer
//  and forward frame's address as callback argument.
class awaiter
{
    // Return might be ignored.
    bool await_suspend(coroutine_handle<> resume_cb)
    {
        int64_t relative_count = -duration.count();
        timer = CreateThreadpoolTimer(TimerCallback,
                                      // Unwrap the handle
                                      resume_cb.address(),
                                      nullptr);
        SetThreadpoolTimer(timer, (PFILETIME)&relative_count, 0, 0);
        return timer != 0;
    }

    // A thread in Windows Thread Pool will invoke this callback.
    //  And it will be resumer for the function.
    static
    void CALLBACK TimerCallback(PTP_CALLBACK_INSTANCE,
                                void *Context,
                                PTP_TIMER)
    {
        // Wrap the frame to handle again. And resume.
        coroutine_handle<>::from_address(Context).resume();
    }
}
```

- `await_resume` : *"Now, here is the return value and control flow"*

Suspend/resume point is created for resume operation. When the coroutine handle's `.resume()` is invoked, the function will set the context(frame pointer) and jump to it. The flow then exits the scope and call `await_resume` .

Notice that we did suspend because the value wasn't ready(`await_ready`). When the function is resumed, there should be return value. If `await_resume` 's return type is void, it means control flow is back.




```
// Resumable Function
future<void> test()
{
    cout << this_thread::get_id() << ": sleeping…\n";

    // await for 1 millisecond...
    co_await 1ms;

    cout << this_thread::get_id() << ": woke up\n";
}
```

`co_await` Tricks

Kenny Kerr and James Mcnellis introduced several tricks with this mechanism. I strongly recommend you to watch the video.

CppCon 2016 : Kenny Kerr & James McNellis “Putting Coroutines to Work with the Windows Runtime”

Async Generator

Concept

- *Async Generator = Awaitable iterator + Resumable function*

This is similar to Generator's concept, but it has **awaitable iterator**. In other words, we can use `for co_await` statement for it. Kirkshoop will give you clear explanation for this.

Example

I won't write hard example since I hadn't try this feature. But previous examples and explanations will help your understanding.

```
// Infinite sequence of interger.
async_generator<int> infinite()
{
    for (int i = 0;; ++i) {
        co_yield i;
    }
}

std::future<void> usecase()
{
    // for co_await statement
    for co_await (int v : infinite())
    {
        // do something with `v`...
    }
}
```

I sad `iterator` is awaitable. So real code of `usecase` will be like following.

```
std::future<void> usecase()
{
    // generator with awaitable(async) iterator
    async_generator<int> gen = infinite();

    for (auto iter = co_await gen.begin();  // `co_await` + `begin()`
         iter != gen.end();     // `end()` has nullptr
         co_await ++iter) // `co_await` + `operator++()`
    {
        int v = *iter;
```

```
            // ...
        }
    }
```

The only difference form `generator<T>` example is that the function is awaiting iterator at 2 points. `begin()` and `operator++()`. These are points for resume operation.

Here, value type is primitive type `int`. So there won't be complex code here. The following code shows how to make awaitable iterator. The other code for generator body and `promise_type` is exactly same with `std::experimental::generator<T>`

```
// Actually, generator is not async. Its iterator is async(awaitable).
template <typename T>
struct async_generator
{
    struct promise_type;    // Same with experimental generator

    // We will implement awaitable interface...
    struct iterator :
            std::iterator<input_iterator_tag, T>
    {
        coroutine_handle<promise_type> chp = nullptr;

        // ...
        iterator operator++(int) = delete;
        iterator &operator++();
        // ...
        T const *operator->() const;
        T const &operator*() const
        {
            // promise_type has constant pointer to value
            const promise_type& prom = chp.promise();
            return *prom.pvalue;
        }
        // ...

        // Since this is syntactic example, there is no suspension.
        bool await_ready() const
        {
            return true;
        }

        // With no suspension, there is nothing to do with handle.
        void await_suspend(coroutine_handle<> hcoro) {}

        // Return awaitable iterator at resume point
        iterator await_resume() const
        {
            return *this;
```

```
        }
    };// iterator

};// async_generator
```

By adding 3 member functions for `iterator`, we could make it awaitable and became available to use `for co_await` statement. But as I mentioned at **Awaitable Interface** section, you don't have to add member function.

```
using iter_type = async_generator<int>::iterator;

bool await_ready(iter_type)
{
    return true;
}

void await_suspend(iter_type & iter,
                   coroutine_handle<> hcoro)
{
    // do for suspension...
}

iter_type& await_resume(iter_type& it)
{
    return it;    // forward the reference
}
```

Adding these helper functions for old types will work fine.

Conclusion

So, we have traveled MSVC coroutine from concept to code. That's all. I wish I made helpful description.

I'd like to add disassembly experience about this feature, but I can't make it because I'm a beginner for that ability. I will try that topic later.

I couldn't make this article in more detail. Maybe there could be a chance to update this...! :D

Blog articles are licensed under a Creative Commons Attribution 4.0 International License.