

System API와 Coroutine 결합하기

박 동 하

C++ Korea Facebook Group

Alchera Inc. 전문연구원

github.com/luncliff

참고자료

표준 관련문서 <https://github.com/cplusplus/draft>

- N4830

영상

- [CppCon 2016: Putting Coroutines to Work with the Windows Runtime](#) – Kenny Kerr & James McNellis

문서

- Linux/FreeBSD Programmer's Manual
- Win32 API: MSDN
- [C++/WinRT: Coroutines and Thread Pool](#) – Kenny Kerr

(권장)입문자료

한글

- C++ Korea 5회 세미나: “C++ Coroutine 알아보기: 접근법, 컴파일러, 그리고 이슈들” (같은 발표자)
 - <https://github.com/CppKorea/CppKoreaSeminar5th>
 - <https://luncliff.github.io/coroutine/ppt/Exploring-the-Cpp-Coroutine/>

영문

- C++ links: Coroutines: by MattPD. <https://gist.github.com/MattPD/9b55db49537a90545a90447392ad3aeb>
- CppCon 2017 : [Gor Nishanov “Naked coroutines live\(with networking\)”](#)
- CppCon 2016 : [Gor Nishanov “C++ Coroutines: Under the covers”](#)
- CppCon 2016 : [James McNellis “Introduction to C++ Coroutines”](#)

발표에서 다루는 것들

C++ Coroutines

- 기초 지식들

System API + Coroutine <https://github.com/luncliff/coroutine>

- **Linux:** eventfd + epoll
- **Windows:** Callback(Win32 Thread Pool)
- **FreeBSD:** Socket operation + kqueue

Conclusion

- Q & A

C++ Coroutines 요약

“Subroutines are special case of ... coroutines”
– Donald Knuth

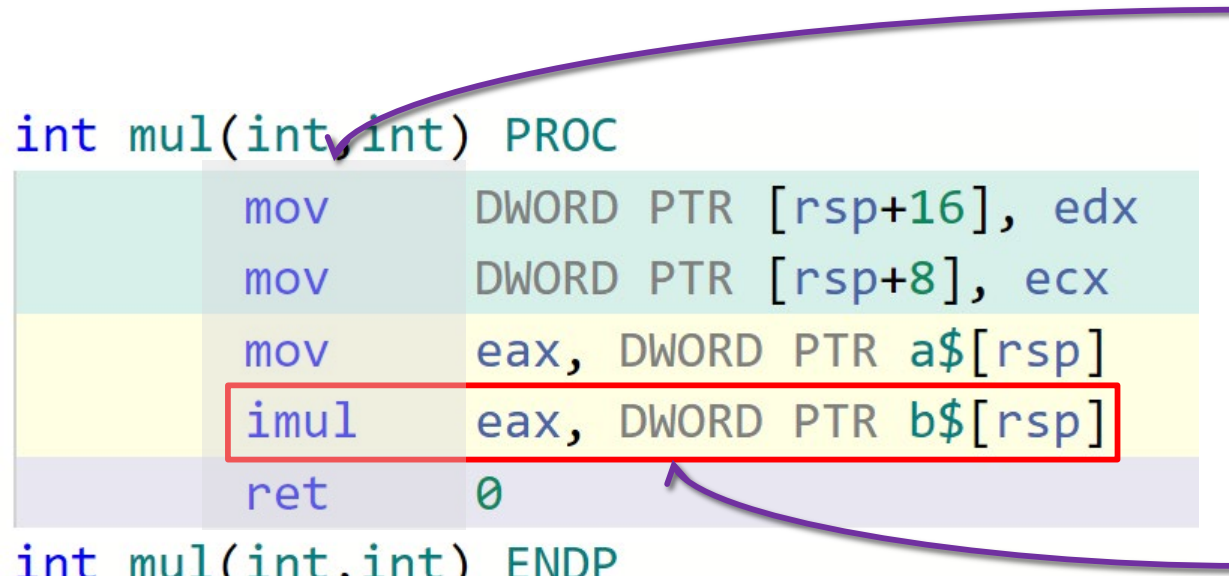


루틴 Routine == 명령 []

루틴 Routine:

- 명령들의 (순서있는) 집합

```
int mul(int, int) PROC
    mov     DWORD PTR [rsp+16], edx
    mov     DWORD PTR [rsp+8], ecx
    mov     eax, DWORD PTR a$[rsp]
    imul    eax, DWORD PTR b$[rsp]
    ret     0
int mul(int, int) ENDP
```



명령 Instruction:

- 기계의 동작 behavior을 추상화 한 것
- 기계 상태가 전이 Transition

루틴 Routine의 상태변화

호출 Invocation

루틴의 **시작점**으로 Jump

활성화 Activation

루틴 **안의 임의 지점**으로 Jump

중단 Suspension

종결하지 않고 다른 루틴의 지점으로 Jump

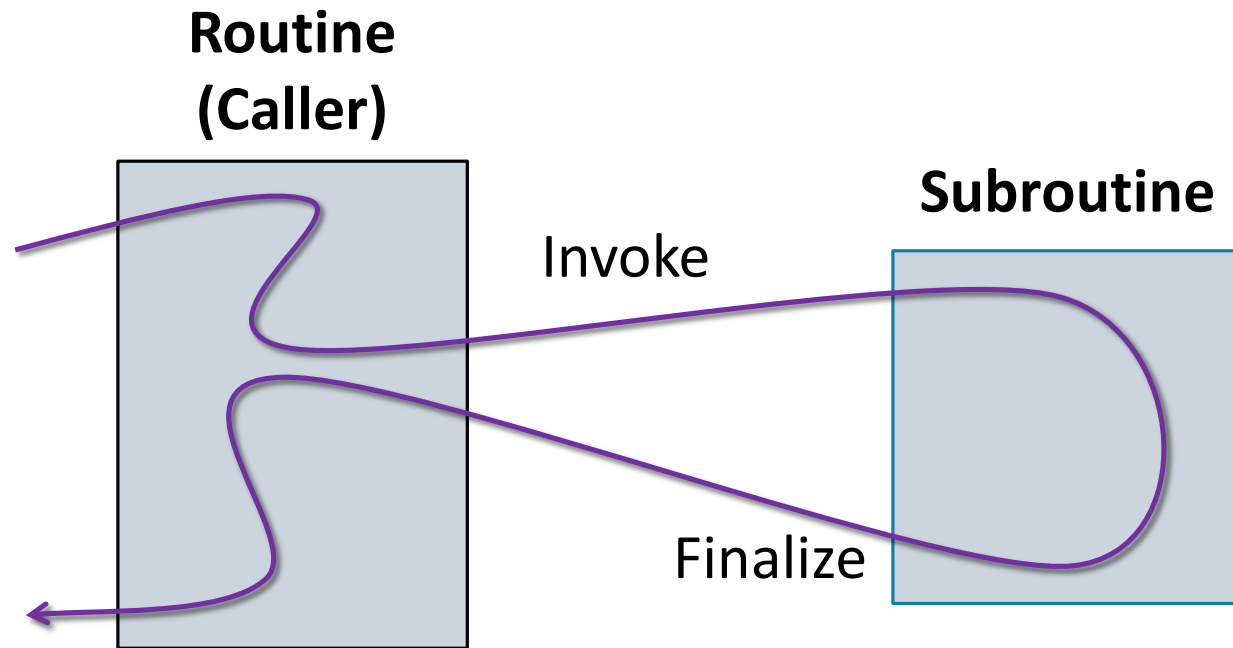
종결 Finalization

루틴의 **끝**에 도달 한 후 루틴 상태의 소멸 및 정리

서브루틴 Subroutine

호출/종결할 수 있는 루틴

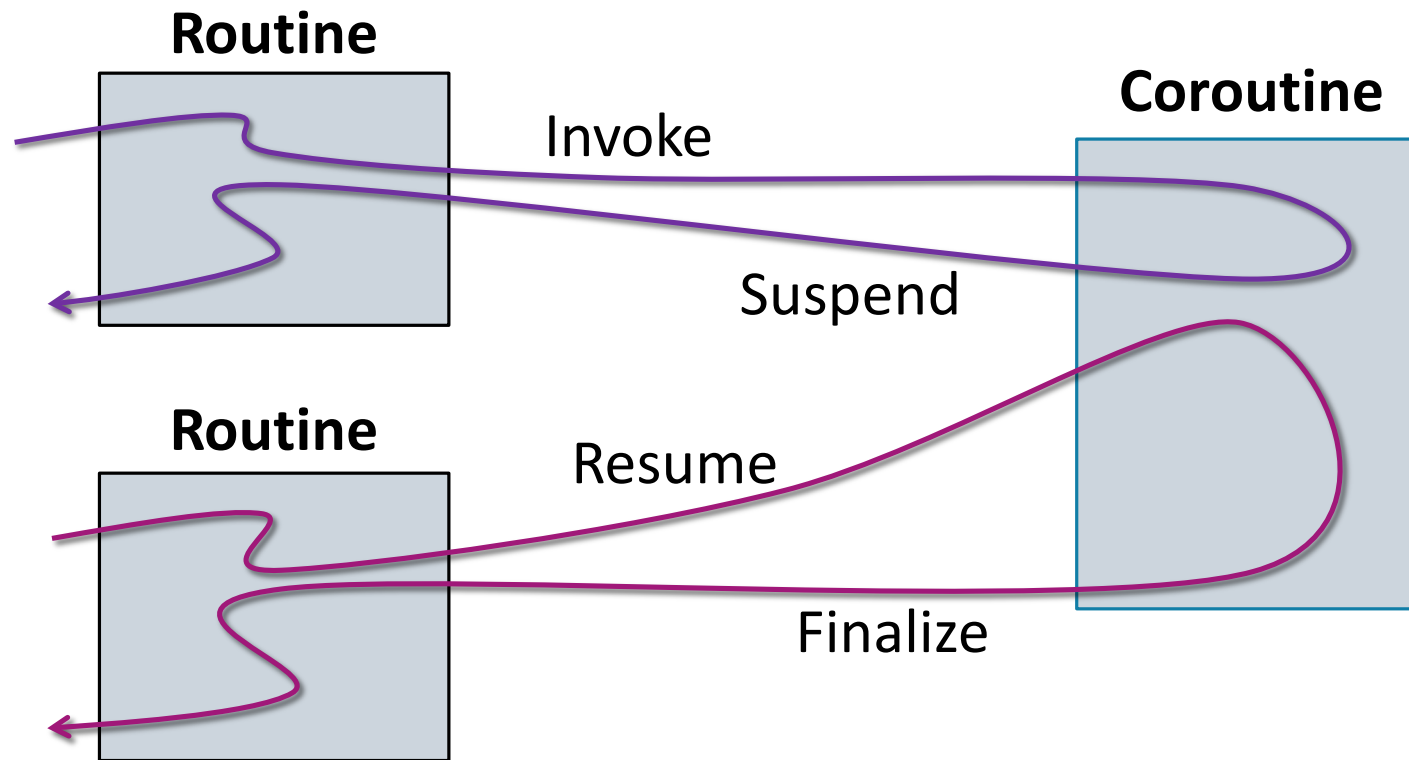
C언어의 모든 함수는 서브루틴



코루틴 Coroutine

호출/종결/중단/재개 할 수 있는 루틴

언어마다 상이한 구현방법을 사용



구현 방법: Stackful & Stackless

호출 스택^{Call Stack}이 있는 경우 코루틴의 구현방법은 2가지

중단한 지점으로 돌아가기 위해서는 함수 프레임이 유지되어야 하기 때문에...
함수 프레임 관리를 어떻게 구현하는지를 통해 구분가능

Stackful

- 함수 프레임을 스택^{Call Stack}에 할당

Stackless

- 함수 프레임을 자유 저장소^{Free Heap Storage}에 할당

C++ Coroutines는 Stackless !!

Stackful 구현 방법을 택한 라이브러리^{Library}들도 존재

컴파일러 지원현황 Compiler Support?

권장 버전 + 컴파일러 옵션

MSVC

- Visual Studio 2017: 15.7.13 (v141) or later
- `/std:c++latest /await`

Clang

- Clang 6 or later
- AppleClang 10 or later
- `-std=c++20 -stdlib=c++ -fcoroutines-ts`

GCC (Work In Progress)

- gcc-10 experimental branch: <https://github.com/iains/gcc-cxx-coroutines>
- `-fcoroutines`

간단히 살펴보면...

개념	C++ Coroutine
----	---------------

호출	변화 없음
----	--------------

종결	<code>co_return</code>
----	------------------------

중단	<code>co_await, co_yield</code> // 1항 연산자 <small>unary operator</small>
----	---

재개	<code>coro.resume()</code> // <code>coroutine_handle<P>::resume()</code>
----	--

C++ Coroutine 은 어떻게 정의 하는가?

함수 안에 다음 중 하나가 존재하면, 그 함수는 코루틴으로 처리한다...

- `co_await` expression
- `co_yield` expression
- `co_return` statement (`co_await/co_yield` 필요)

Coroutine 코드의 구성요소들Components

Awaitable

`co_await`의 피연산자

- `await_ready`
- `await_suspend`, `await_resume`

중단Suspension 제어

Promise

Coroutine 코드 생성

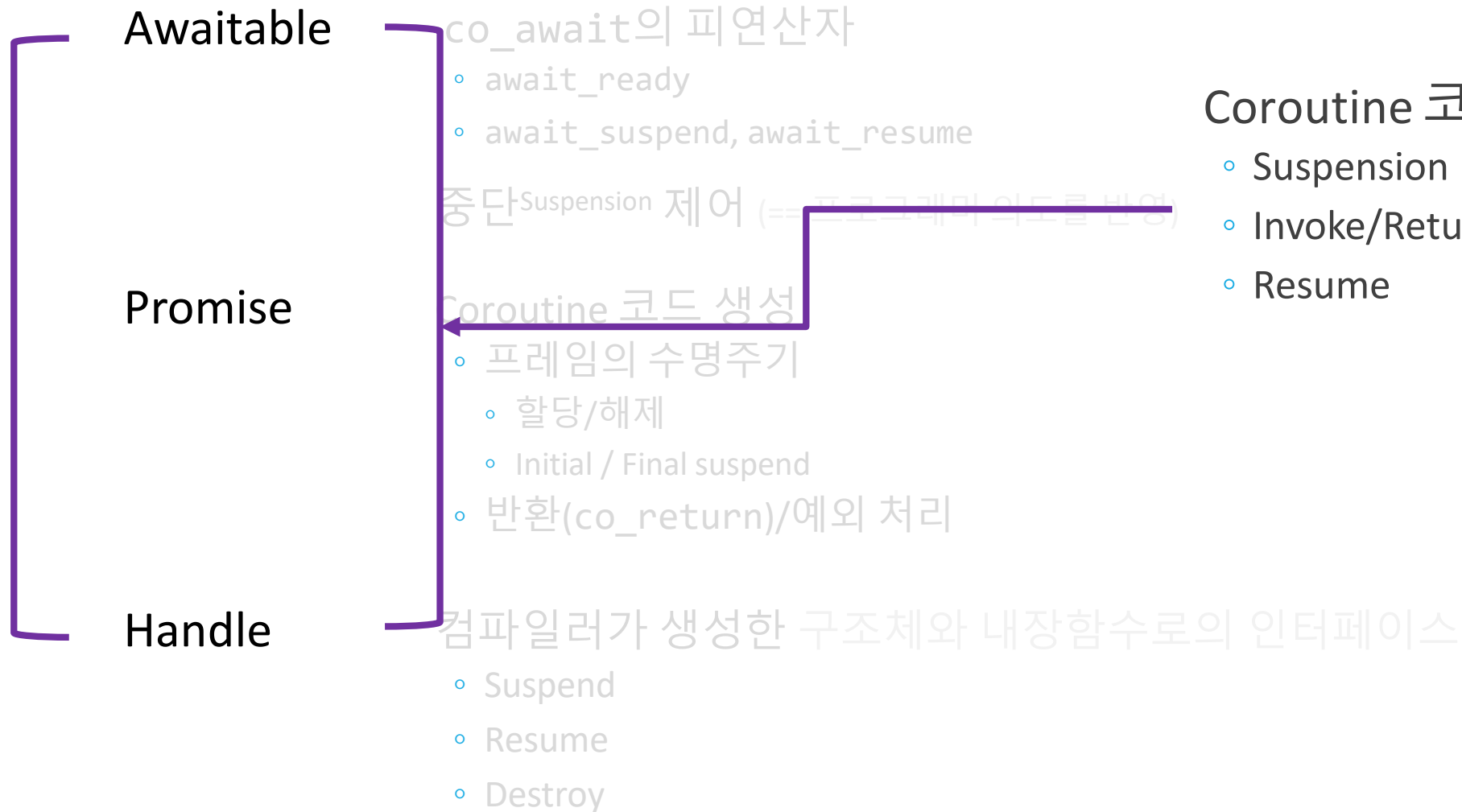
- 프레임의 수명주기
 - 할당/해제
 - Initial / Final suspend
- 반환(`co_return`)/예외 처리

Handle

컴파일러가 생성한 구조체와 내장함수로의 인터페이스

- Suspend
- Resume
- Destroy

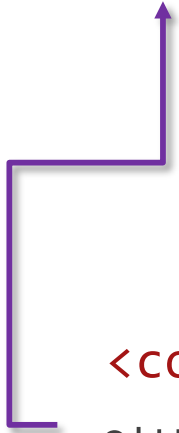
Coroutine 코드의 구성요소들Components



Coroutine 구성요소: Awaitable

Operand for `co_await`


```
#include <experimental/coroutine> // C++17 (Coroutines TS)  
#include <coroutine> // C++ 20
```



<coroutine> 구현은 아직 현재 진행중...

이번 발표(2019.09)에서는 <experimental/coroutine> 을 사용합니다.

헤더파일

```
// #include <experimental/coroutine>
// namespace std::experimental

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

struct suspend_always
{
    bool await_ready() {
        return false;
    }
    void await_suspend(coroutine_handle<void>){}
    void await_resume(){}
};
```

미리 정의된 Awaitable 타입들

```
// #include <experimental/coroutine>
// namespace std::experimental

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

struct suspend_always
{
    bool await_ready() {
        return false;
    }
    void await_suspend(coroutine_handle<void>){}
    void await_resume(){}
};
```

3개의 Member Function이 있으면
Awaitable 타입이 될 수 있다

```
// #include <experimental/coroutine>
```

```
// namespace std::experimental
```

```
struct suspend_never
```

```
{
```

```
    bool await_ready() {
```

```
        return true;
```

```
    }
```

```
    void await_suspend(coroutine_handle<void>) {}
```

```
    void await_resume() {}
```

```
};
```

```
struct suspend_always
```

```
{
```

```
    bool await_ready() {
```

```
        return false;
```

```
    }
```

```
    void await_suspend(coroutine_handle<void>){}
```

```
    void await_resume(){}
```

```
};
```

중단 여부(**bool**)를 결정

중단하는 경우,
coroutine_handle<void> 획득

마지막 중단점 Suspend Point 으로 복귀 Resume

각 **await_** 함수들의 역할

```
#include <experimental/coroutine>
using namespace std::experimental;

auto example_use_await(awaitable& aw) -> forget {
    co_await aw;
}
```

사용자가 `co_await`을 사용하면...

```
#include <experimental/coroutine>
using namespace std::experimental;

auto example_use_await(awaitable& aw) -> forget {
    // ...
    if (aw.await_ready() == false) {
        auto handle = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(handle);
        // ... update suspend point + return ...
    }
    __suspend_point_n:
    aw.await_resume();
    // ...
}
```

컴파일러의 코드 생성

```
#include <experimental/coroutine>
using namespace std::experimental;
```

```
auto example_use_await(awaitable& aw) -> forget {
    // ...
    if (aw.await_ready() == false) {
        auto handle = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(handle);
        // ... update suspend point + return ...
    }
    __suspend_point_n:
    aw.await_resume();
    // ...
}
```

중단 여부(**bool**)를 결정

중단하는 경우,
coroutine_handle<void> 획득

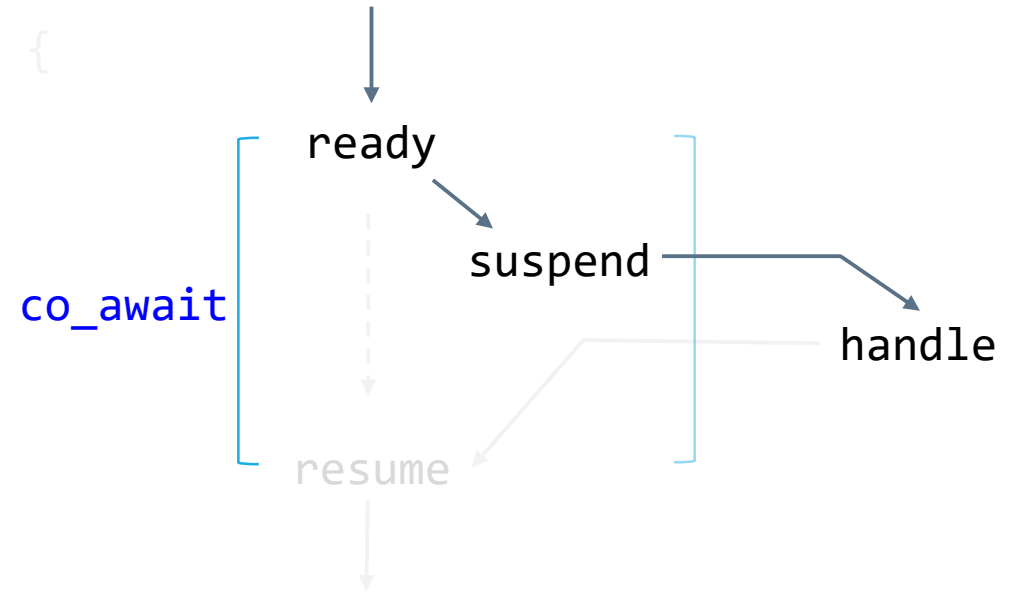
마지막 중단점 Suspend Point으로 복귀 Resume

await_ 함수 == Callback

```

auto example_use_await(suspend_always& aw) -> forget {
    // ...
    if (aw.await_ready() == false) {
        auto handle = coroutine_handle<promise
        aw.await_suspend(handle);
        // ... update suspend point + return ...
    }
    __suspend_point_n:
    aw.await_resume();
    // ...
}

```

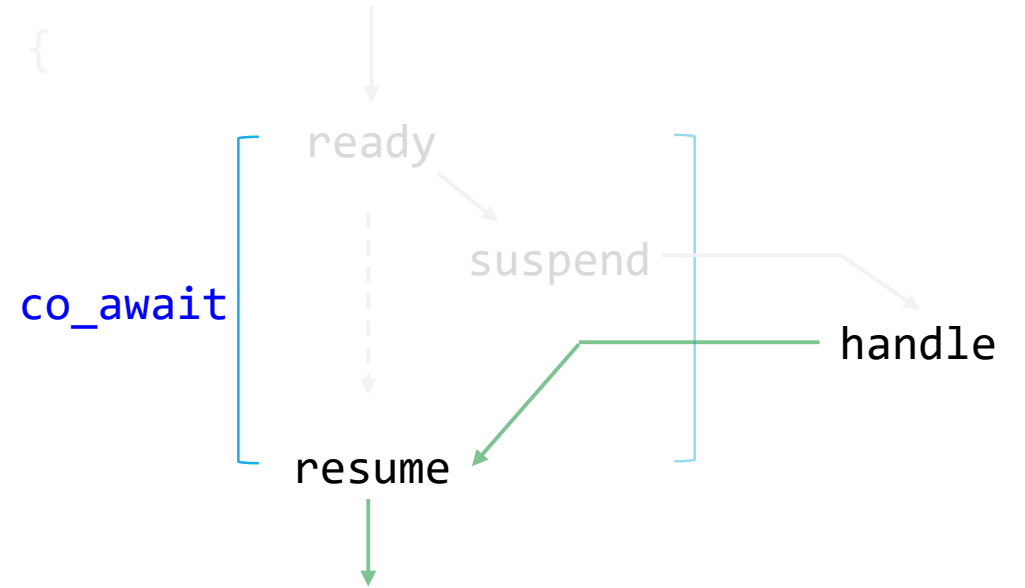


Suspension
`await_ready() == false`


```

auto example_use_await(suspend_always& aw) -> forget {
    // ...
    if (aw.await_ready() == false) {
        auto handle = coroutine_handle<promise
        aw.await_suspend(handle);
        // ... update suspend point + return ...
    }
    __suspend_point_n:
    aw.await_resume();
    // ...
}

```

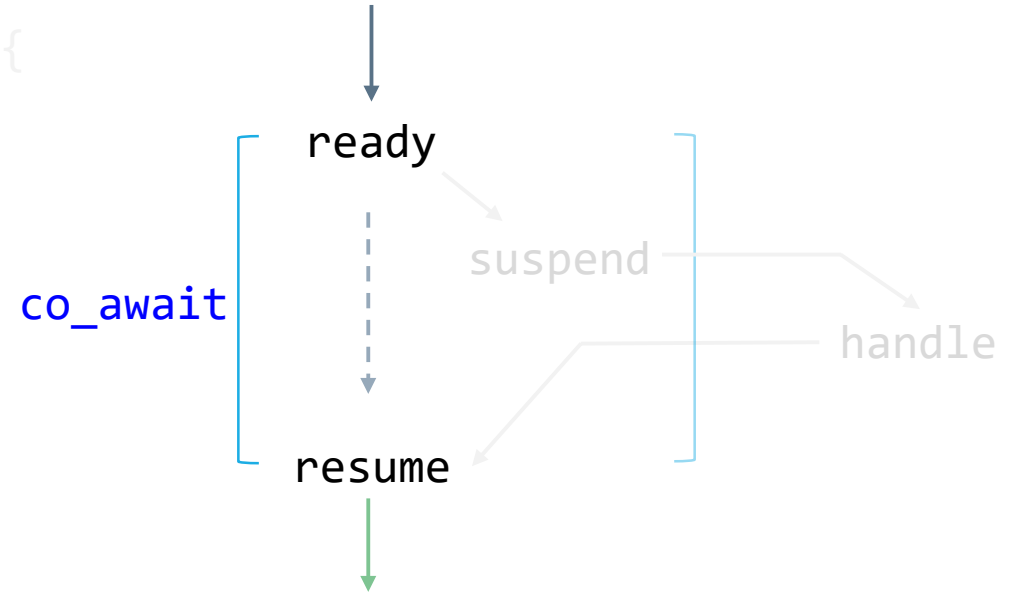


Resume
`await_ready() == false`

```

auto example_use_await(suspend_never& aw) -> forget {
    // ...
    if (aw.await_ready() == false) {
        auto handle = coroutine_handle<
        aw.await_suspend(handle);
        // ... update suspend point + return ...
    }
    __suspend_point_n:
    aw.await_resume();
    // ...
}

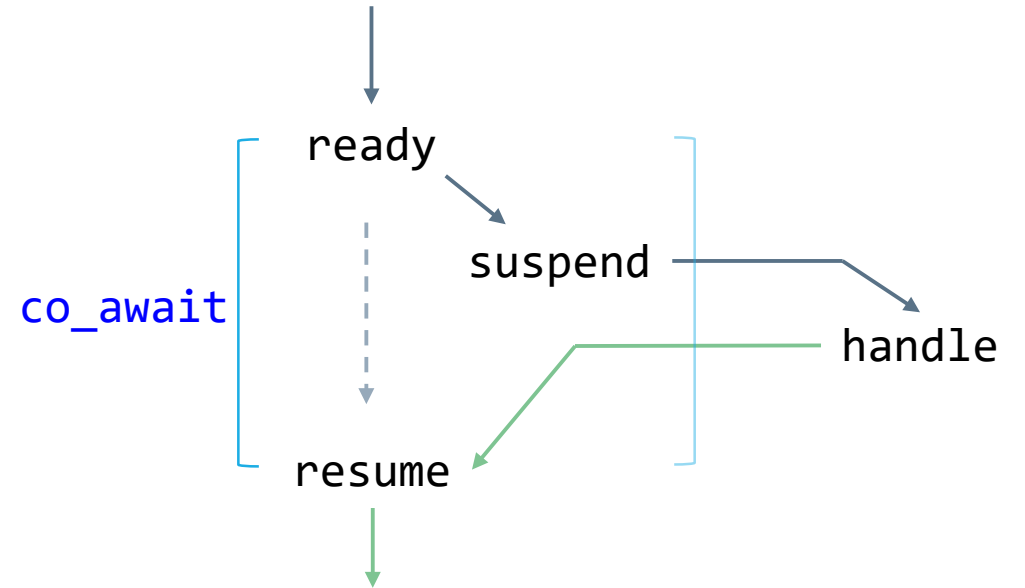
```



Bypass
`await_ready() == true`

```
#include <experimental/coroutine>
using namespace std::experimental;

auto example_use_await(awaitable& aw) -> forget {
    co_await aw;
}
```



그림으로 표현한 `co_await`

Coroutine 구성요소: Promise

Type System을 통한 Coroutine Frame관리

Promise Type: 가장 많은 일을 하는 Component

반환 처리

- `co_return`: return_void, return_value
- `co_yield`: yield_value

Coroutine의 ReturnType 생성

- `get_return_object`, `get_return_object_on_allocation_failure`

예외처리

- `unhandled_exception`

Frame 생성/소멸

- Operator `new/delete`

예약 중단점

- `initial_suspend` : 호출 직후 프로그래머의 코드로 진입하는가?
- `final_suspend` : 반환 이후 코루틴의 함수 프레임을 파괴하는가?

Promise Type: 필수사항

반환 처리

- `co_return`: `return_void`, `return_value`
- `co_yield`: `yield_value`

Coroutine의 ReturnType 생성

- `get_return_object`, `get_return_object_on_allocation_failure`

예외처리

- `unhandled_exception`

Frame 생성/소멸

- `Operator new/delete`

예약 중단점

- `initial_suspend`
- `final_suspend`

```
using namespace std::experimental;
auto example() -> return_type
{
    // ... programmer's code ...
}
```

코루틴을 작성하면...

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };

    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
    __final_suspend_point:
    co_await p.final_suspend();
}
```

Promise를 통한 코드 생성


```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };

    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
    __final_suspend_point:
    co_await p.final_suspend();
}
```

내 코드는 어디에?

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };

    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
    __final_suspend_point:
    co_await p.final_suspend();
}
```

대부분이 `promise_type`의 멤버함수

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };

    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
    __final_suspend_point:
    co_await p.final_suspend();
}
```

Coroutine Body Behavior

```
#include <experimental/coroutine>
using namespace std::experimental;
```

```
auto example() -> forget {
    // ...
    co_await suspend_never{};
    co_return;
    // ...
}
```

```
struct forget {
    struct promise_type {
        suspend_never initial_suspend() { return {}; }
        suspend_never final_suspend() { return {}; }

        void return_void(){}

        auto get_return_object() -> forget {
            return {this};
        }
    };

    forget(const promise_type*) {}
};
```

forget 의 정의

```
#include <experimental/coroutine>
using namespace std::experimental;
```

```
auto example() -> forget {
    // ...
    co_await suspend_never{};
    co_return;
    // ...
}
```

```
struct forget {
    struct promise_type {
        suspend_never initial_suspend() { return {}; }
        suspend_never final_suspend() { return {}; }

        void return_void(){}

        auto get_return_object() -> forget {
            return {this};
        }
    };

    forget(const promise_type*) {}
};
```

Return개체 생성

```
#include <experimental/coroutine>
using namespace std::experimental;
```

```
auto example() -> forget {
    // ...
    co_await suspend_never{};
    co_return;
    // ...
}
```

```
struct forget {
    struct promise_type {
        suspend_never initial_suspend() { return {}; }
        suspend_never final_suspend() { return {}; }

        void return_void(){}

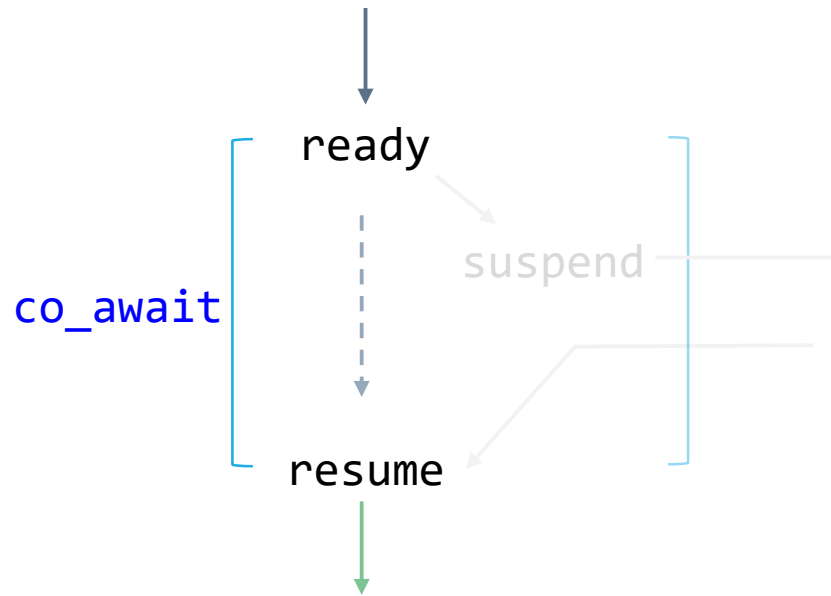
        auto get_return_object() -> forget {
            return {this};
        }
    };

    forget(const promise_type*) {}
};
```

Return 처리

```
#include <experimental/coroutine>
using namespace std::experimental;
```

```
auto example() -> forget {
    // co_await initial_suspend();
    co_await suspend_never{};
    co_return;
    // ...
}
```



```
struct forget {
    struct promise_type {
        suspend_never initial_suspend() { return {}; }
        suspend_never final_suspend() { return {}; }

        void return_void(){}

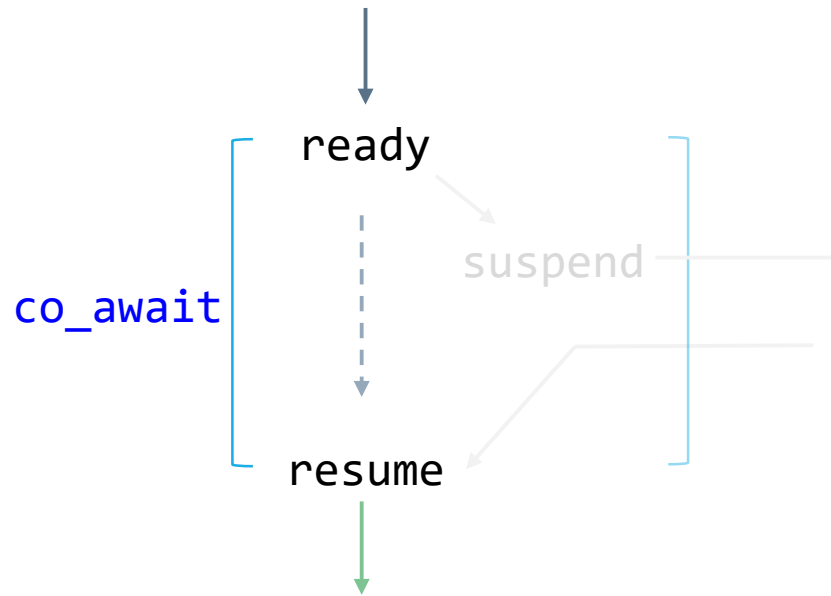
        auto get_return_object() -> forget {
            return {this};
        }
    };

    forget(const promise_type*) {}
};
```

호출 후 Initial 중단 Suspend?

```
#include <experimental/coroutine>
using namespace std::experimental;
```

```
auto example() -> forget {
    // co_await initial_suspend();
    co_await suspend_never{};
    co_return;
    // co_await final_suspend();
}
```



```
struct forget {
    struct promise_type {
        suspend_never initial_suspend() { return {}; }
        suspend_never final_suspend() { return {}; }

        void return_void(){}

        auto get_return_object() -> forget {
            return {this};
        }
    };

    forget(const promise_type*) {}
};
```

반환 후^{final} 중단^{Suspend?}


```
#include <experimental/coroutine>
using namespace std::experimental;
```

```
auto example() -> forget {
    // ...
    co_await suspend_never{};
    co_return;
    // ...
}
```

```
struct forget {
    struct promise_type {
        suspend_never initial_suspend() { return {}; }
        suspend_never final_suspend() { return {}; }

        void return_void(){}

        auto get_return_object() -> forget {
            return {this};
        }
    };

    forget(const promise_type*) {}
};
```

즉, `forget` == Coroutine을 위한 `void`

Coroutine 구성요소: Handle

Interface for coroutine frame/operations

Awaitable과 Promise의 역할을 정리해보면...

Awaitable

- 중단^{Suspension}을 위한 연산자 `co_await`

Promise

- 프레임^{Frame} / 예외^{Exception} 관리: 이번 발표에서는 다루지 않음
- 반환^{Finalization} 처리: `co_return`

호출^{Invoke}은 Subroutine과 완전히 동일...

- 재개^{Resume}는 어떻게 하는거지?
- Final Suspend 했다면 어떻게 소멸^{Destruction}시키지?

Coroutine Handle의 역할

Library 형태로 Coroutine Frame을 위한 안전한^{Safe} 사용법^{Interface}

- Observer : `done()`
- Resume : `resume()`
- Destruction : `destroy()`
- Promise Access : `coroutine_handle<promise_type>`

```
template <typename PromiseType = void>
struct coroutine_handle;
```

Template 으로 다양한 타입을 지원

```
template <>
struct coroutine_handle<void>
```

```
{
```

Coroutine(Stack-less) Frame 주소

```
private:
```

```
void* __handle__;
```

마지막 중단 지점으로 Resume

- `co_await`의 `await_resume`으로 연결

```
public:
```

```
operator bool() const;
```

```
void operator()();
```

Frame 소멸처리

`co_return` 여부 검사

```
void resume();
```

```
void destroy();
```

```
bool done() const;
```

`void*` 와 상호 변환 가능

```
void* address() const;
```

```
static coroutine_handle from_address(void*);
```

```
};
```

```
void resume_until_done(coroutine_handle<void>& coro) // ... get handle somehow ...
{
    if (coro.address() == nullptr)
        return;

    while (coro.done() == false){
        coro.resume()
    }

    return coro.destroy();
}
```

종료 Return 확인 / 재개 Resume

Frame 소멸 시도

coroutine_handle<void> 사용 예시

```

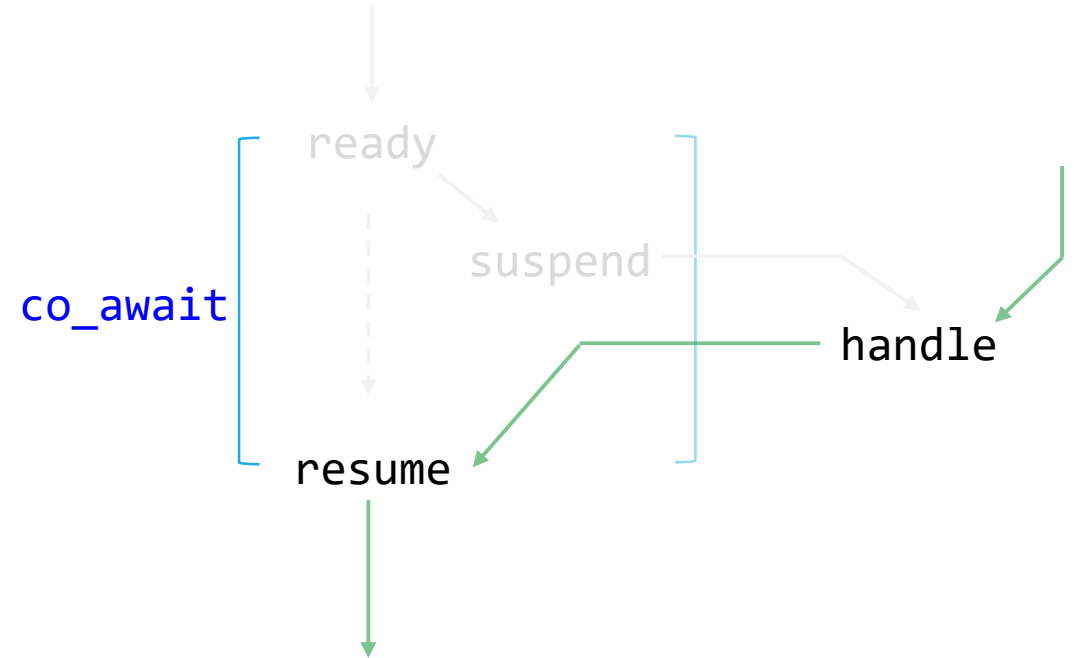
void resume_until_done(coroutine_handle<void>& coro)
{
    while (coro.done() == false){
        coro.resume()
    }
    return coro.destroy();
}

```

```

auto example_use_await(suspend_always& aw)
// ...
if (aw.await_ready() == false) {
    auto handle = coroutine_handle<promise
    aw.await_suspend(handle);
    // ... update suspend point + return ...
}
__suspend_point_n:
    aw.await_resume();
// ...
}

```



Resume: 마지막 중단점 Suspend Point 으로
goto

```

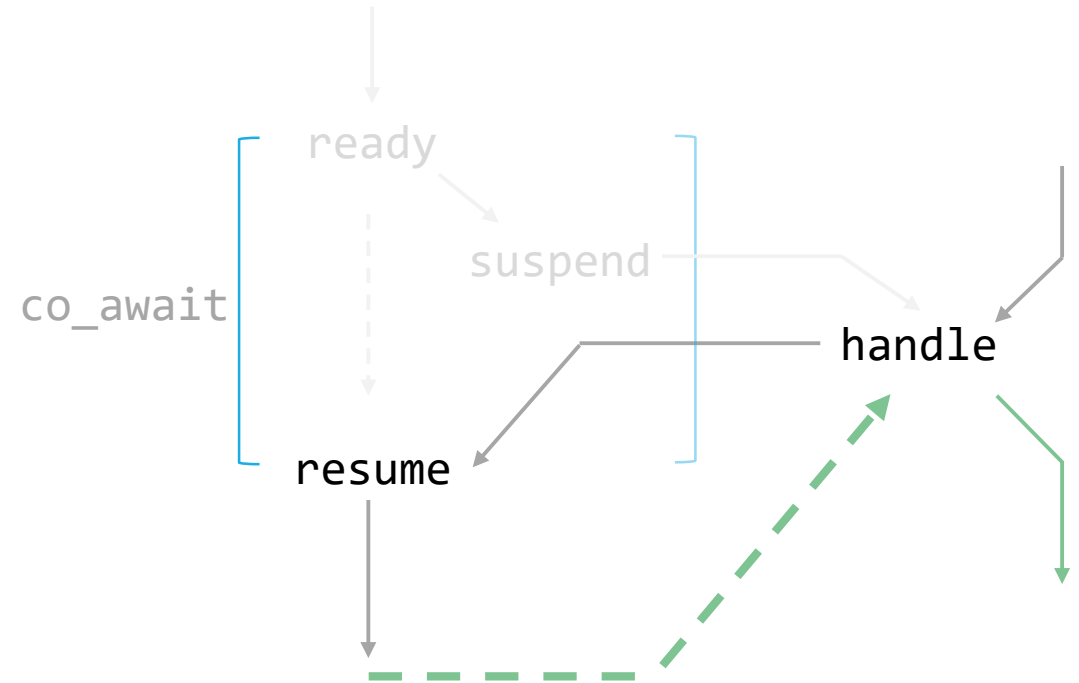
void resume_until_done(coroutine_handle<void>& coro)
{
    while (coro.done() == false){
        coro.resume()
    }
    return coro.destroy();
}

```

```

auto example_use_await(suspend_always& aw)
// ...
if (aw.await_ready() == false) {
    auto handle = coroutine_handle<promise
    aw.await_suspend(handle);
    // ... update suspend point + return ...
}
__suspend_point_n:
    aw.await_resume();
    // ... final suspension + destruction ...
}

```



Coroutine의 중단Suspend/종결Finalization


```

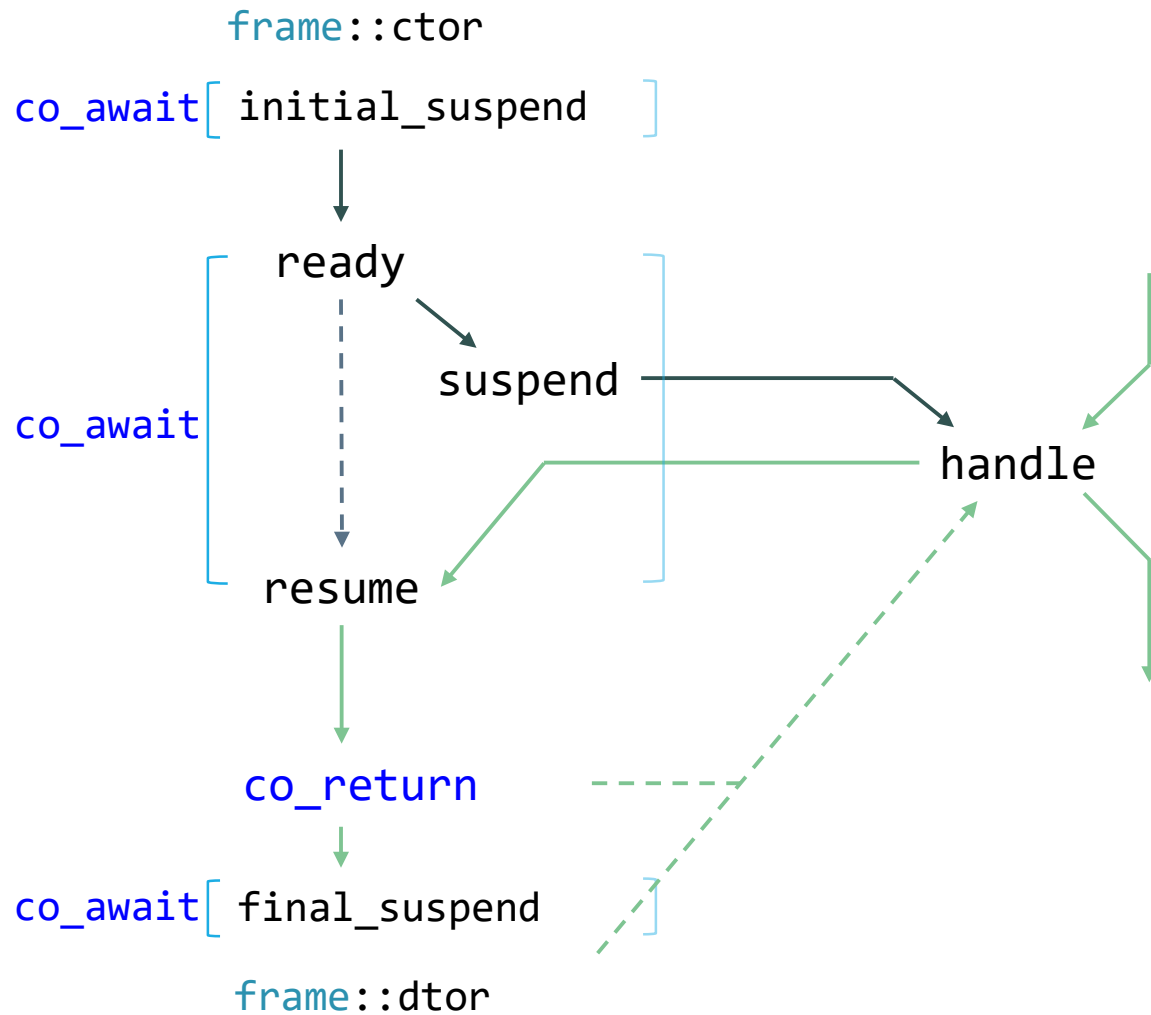
void resume_until_done(coroutine_handle<void>& coro) // ... get handle some how ...
{
    while (coro.done() == false){
        coro.resume()
    }
    return coro.destroy();
}

class some_coroutine_return_type {
    class promise_type {
    public:
        auto final_suspend() -> some_awaitable_type {
            return {};
        }
        // ...
    };
};

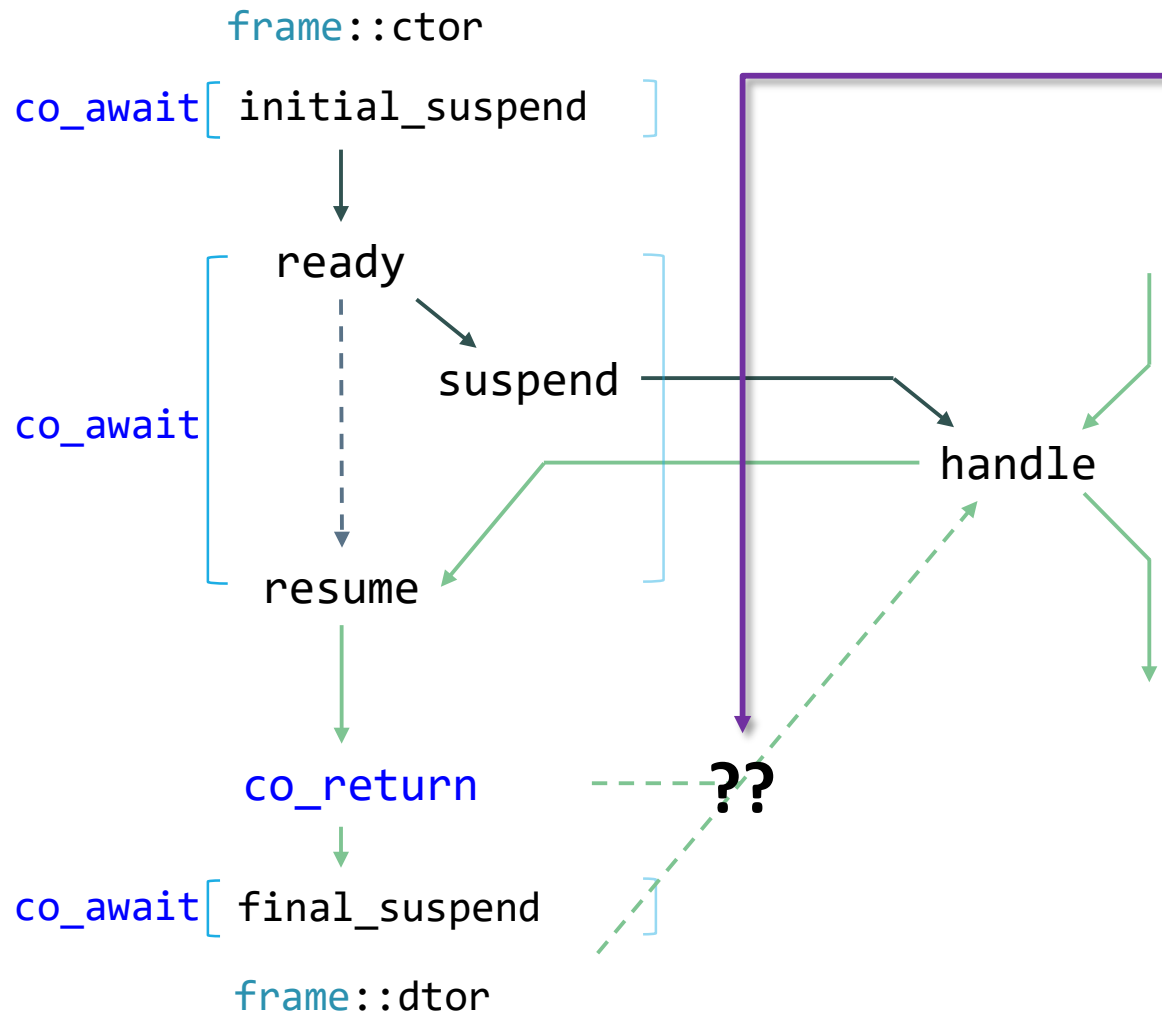
```

Q: 정말 소멸^{Destruction} 해도 괜찮은가?

A: Final Suspend에 따라 달라진다



좀 더 구체적으로 그린
The shape of the Coroutine



- `if final_suspend() -> suspend_always`
 - Coroutine은 중단된 상태 == Safe for Destruction
- `if final_suspend() -> suspend_never`
 - Frame은 이미 소멸됨 == Double Delete

Destroy 해도 좋은가?
Handle을 사용하는 쪽에서는 알 수 없다!

```
void resume_until_done(coroutine_handle<void>& coro) // ... get handle some how ...
{
    while (coro.done() == false){
        coro.resume()
    }
    return coro.destroy();
}
```

Coroutine의 현재 상태를 고려하지 않음

- Frame 내 변수들(+ Promise)이 이미 소멸했을 수 있다!
- 프로그래머가 사려깊게 설계해야하는 부분

Linux: Event Polling

Suspending for an event + Resume with a loop

목적 Motivation

Event Notification을 `co_await`하고싶다!

```
auto event_wait_coroutine(auto_reset_event& e) -> forget {  
    co_await e;  
    // ... resume after the event is signaled ...  
    // ...  
    co_await e;  
    // ...  
}
```

```
void event_signal_subroutine(auto_reset_event& e){  
    // ... signal when something happens ...  
    e.set();  
}
```

분석Analysis

Linux System API와 관련해서 확인해야하는 내용들

- Linux Realtime Extension
 - eventfd
- I/O Multiplexing
 - epoll

eventfd

File-Like interface를 가진 Event 개체

- Event 통지/수신처리에 File I/O 함수를 사용
- `write` == Notification
- `read` == Wait

특징

- System 의존적
- I/O의 길이가 고정적: `uint64_t`
- Write의 누적 + 누적 한계값 Sentinel Value : `0xffff'ffff'ffff'fffe`

```
#include <sys/eventfd.h>
```

```
int eventfd(unsigned int initval, int flags);
```

<http://man7.org/linux/man-pages/man2/eventfd.2.html>

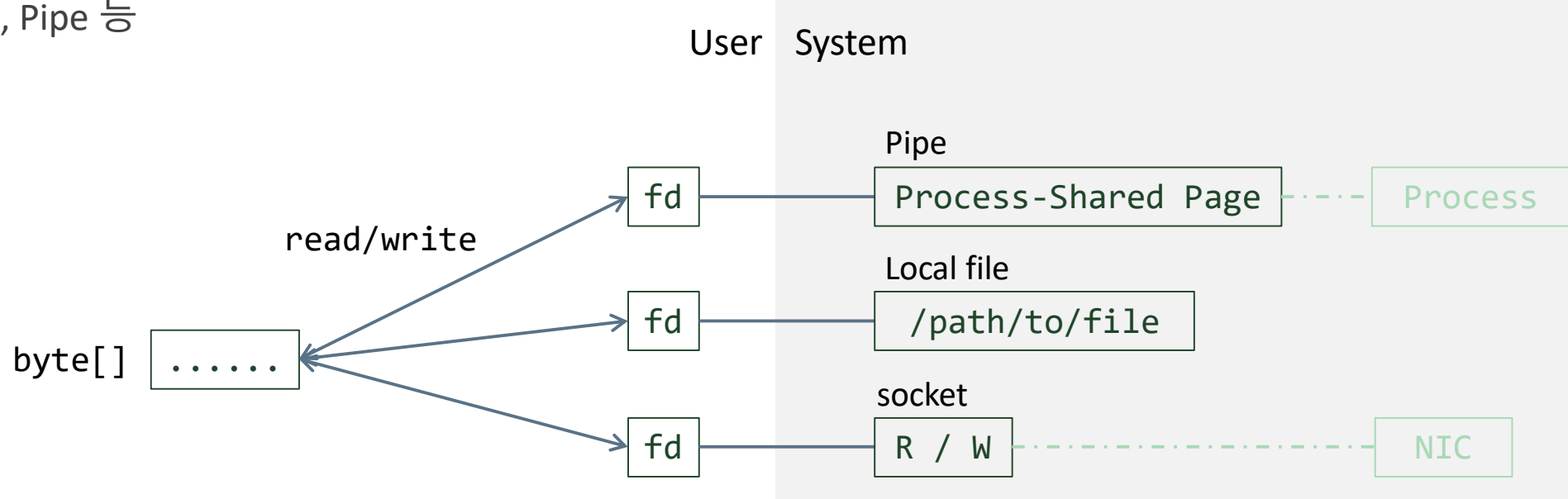
File Descriptor

Kernel Object로 접근할 때 사용하는 Handle

- 일반적으로 `int` (언어/플랫폼에 따라서는 `uintptr`) 사용
- 명시적인 `close` 처리

File-Like?

- 발표자가 편의상 사용하는 용어 (보통은 File 혹은 FD라고 표현)
- 입출력 I/O이 가능한 개체^{Object}를 통칭: `read/write` 가능
- Local File, Socket, Pipe 등



Linux Realtime Extension

Linux 시스템에서는 보다 범용으로 쓰일 수 있는 FD 제공

Realtime이라는 이름처럼 실행시간에 적절한 처리가 필요한 부분들과 관련됨

- `signalfd` : OS Signal 수신
- `eventfd` : Event 발생을 통지^{Notification}
- `timerfd` : Interval 측정과 같은 Timer 기능

Q. 왜 하필 File-Like 형태로 지원하는걸까?

I/O Multiplexing

목표: 많은 I/O Context를 효율적으로 처리하기 위한 기능

- https://en.wikipedia.org/wiki/C10k_problem

대상: 제어 흐름 Control Flow

- 단일 프로세서(보통 Thread를 의미)에서 동시적으로 처리

대표적으로 POSIX select/poll

- `#include <sys/select.h>`
- `#include <poll.h>`

```

#include <poll.h>

errc example(struct pollfd* fds, size_t capacity, int millisec = 500){
    auto count = poll(fds, 2, millisec);
    if (count <= 0) // timeout(0) or error(-)
        return errc{errno};

    for_each(fds, fds + capacity, [&count](struct pollfd& p) {
        if (p.revents & POLLWRBAND) {
            // ...
        }
        if (p.revents & POLLOUT) {
            // ...
        }
        if (p.revents & POLLHUP) {
            // ...
        }
    });
    return errc{};
}

```

Pattern: Examine then Process

```
#include <poll.h>
```

```
errc example(struct pollfd* fds, size_t capacity, int millisec = 500){  
    auto count = poll(fds, 2, millisec);  
    if (count <= 0) // timeout(0) or error(-)  
        return errc{errno};  
  
    for_each(fds, fds + capacity, [&count](struct pollfd& p) {  
        if (p.revents & POLLWRBAND) {  
            // ...  
        }  
        if (p.revents & POLLOUT) {  
            // ...  
        }  
        if (p.revents & POLLHUP) {  
            // ...  
        }  
    });  
    return errc{};  
}
```

1. FD Group을 생성 (Focus)

```
#include <poll.h>
```

```
errc example(struct pollfd* fds, size_t capacity, int millisec = 500){
```

```
    auto count = poll(fds, 2, millisec);
```

```
    if (count <= 0) // timeout(0) or error(-)
```

```
        return errc{errno};
```

1. FD Group을 생성 (Focus)

```
    for_each(fds, fds + capacity, [&count](struct pollfd& p) {
```

```
        if (p.revents & POLLWRBAND) {
```

```
            // ...
```

```
        }
```

```
        if (p.revents & POLLOUT) {
```

```
            // ...
```

```
        }
```

```
        if (p.revents & POLLHUP) {
```

```
            // ...
```

```
        }
```

```
    });
```

```
    return errc{};
```

```
}
```

2. FD Group의 상태를 확인 (Examine)

```
#include <poll.h>
```

```
errc example(struct pollfd* fds, size_t capacity, int millisec = 500){
```

```
    auto count = poll(fds, 2, millisec);
```

```
    if (count <= 0) // timeout(0) or error(-)
```

```
        return errc{errno};
```

1. FD Group을 생성 (Focus)

```
    for_each(fds, fds + capacity, [&count](struct pollfd& p) {
```

```
        if (p.revents & POLLWRBAND) {
```

```
            // ...
```

```
        }
```

```
        if (p.revents & POLLOUT) {
```

```
            // ...
```

```
        }
```

```
        if (p.revents & POLLHUP) {
```

```
            // ...
```

```
        }
```

```
    });
```

```
    return errc{};
```

```
}
```

2. FD Group의 상태를 확인 (Examine)

3. 각각에 맞는 처리를 수행 (Process)

epoll

Linux-specific I/O Event Notification

- 목적, 패턴 모두 POSIX `select`, `poll`과 거의 같음

User Space에서 소모되는 시간을 최소화

- FD Group을 직접 관리하지 않고 Epoll-Object에 Registration 하는 방식

<http://man7.org/linux/man-pages/man7/epoll.7.html>


```
#include <sys/epoll.h>
```

Epoll Object 생성

```
int epoll_create1(int flags);
```

Epoll에 FD를 등록

```
int epoll_ctl(int epfd, int op,  
              int fd, struct epoll_event* event);
```

```
struct epoll_event {  
    uint32_t events; // epoll events  
    epoll_data_t data; // user's data  
};
```

Epoll을 사용한 이유:

```
typedef union epoll_data {
```

`coroutine_handle<void> == void*`

```
    void* ptr;  
    int fd;  
    uint32_t u32;  
    uint64_t u64;  
} epoll_data_t;
```

누적된 Event 확인

```
int epoll_wait(int epfd, struct epoll_event* events, int maxevents,  
               int timeout);
```

epoll 관련 함수들

설계^{Design}: Awaitable

Event Object + RAII

- 생성자: `eventfd`
- 소멸자: `close(fd)`

Ready

- Notification 상태에 따라서 분기

Suspend

- `coroutine_handle<void>`를 `epoll_data_t`로 Epoll에 등록
- 1회 Notification 되면 등록을 해제

Resume

- Signal 상태를 Reset 처리: `auto_reset_event` (계속적으로 사용가능)
- 1 Coroutine == 1 Event Object

설계^{Design}: Handle

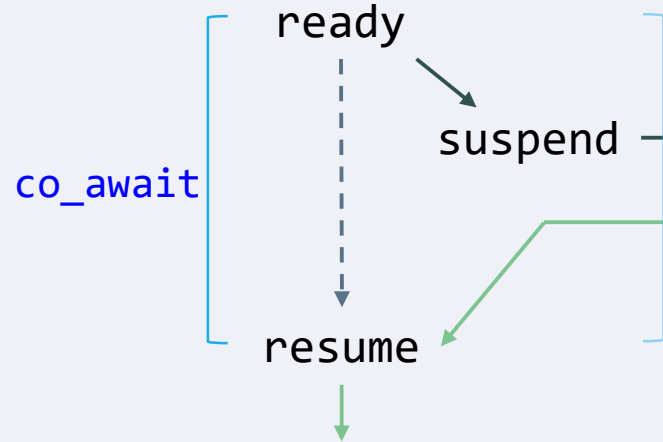
Event Polling을 수행하는 함수를 하나만 사용하도록 강제(유일한 사용법)

Resume 가능한 Coroutine들을 일제히 반환

```
#include <experimental/generator>
auto signaled_event_tasks() -> generator<coroutine_handle<void>>;

auto resume_event_waiting_coroutines() {
    // query and resume
    // when there are signaled coroutines
    for (auto coro : signaled_event_tasks())
        coro.resume();
}
```

Awaitable



Handle



signaled_event_tasks: epoll_wait

auto_reset_event

: eventfd

◦ await_ready

◦ await_suspend : epoll_ctl

◦ await_resume : read

3rd Routine

◦ Event Notification : write

API 배치

```
class auto_reset_event {
    uint64_t state;

public:
    auto_reset_event(); // eventfd
    ~auto_reset_event(); // close

    bool await_ready(); // bit masking
    void set();          // set state: write
    void await_resume(); // reset state: read

    void await_suspend(coroutine_handle<void>); // epoll_ctl
};
```

Overview: Awaitable Event

```

class auto_reset_event {
    uint64_t state;

public:
    auto_reset_event(); // eventfd
    ~auto_reset_event(); // close
};

auto_reset_event::auto_reset_event() : state{} {
    auto fd = eventfd(0, EFD_NONBLOCK);
    if (fd == -1)
        throw system_error{errno, system_category(), "eventfd"};

    this->state = fd; // start with unsigned state
}

auto_reset_event::~~auto_reset_event() {
    // prevent invalid argument
    auto fd = get_eventfd(this->state);
    close(fd);
}

```

생성자^{Ctor} / 소멸자^{Dtor}

```
class auto_reset_event {
    uint64_t state;

public:
    bool await_ready() {
        return is_signaled(this->state);
    }
};

constexpr uint64_t emask = 1ULL << 63;

bool is_signaled(uint64_t state) {
    return emask & state; // msb is 1?
}

int64_t get_eventfd(uint64_t state) {
    return static_cast<int64_t>(~emask & state);
}
```

Ready: Bit masking

```

class auto_reset_event {
    uint64_t state;

public:
    bool await_ready(); // bit masking
    void set();          // set state: write
};

void auto_reset_event::set() {
    if (is_signaled(this->state)) // already signaled
        return;

    auto fd = get_eventfd(this->state);
    notify_event(fd);
    this->state = emask | static_cast<uint64_t>(fd);
}

void notify_event(int64_t efd) noexcept(false) {
    if (write(efd, &efd, sizeof(efd)) == -1)
        throw system_error{errno, system_category(), "write"};
}

```

Event Set


```

class auto_reset_event {
    uint64_t state;

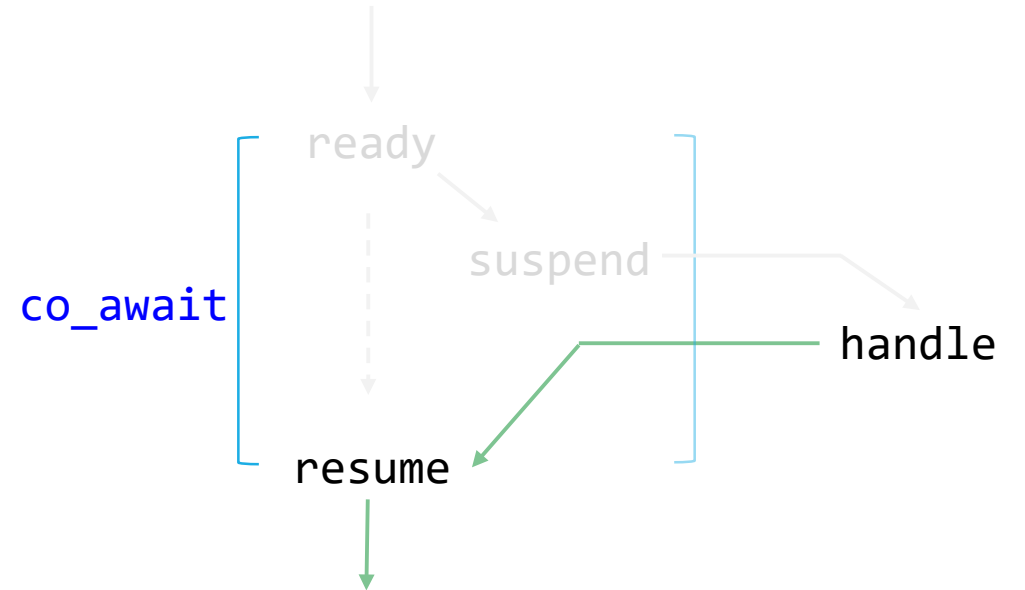
public:
    bool await_ready(); // bit masking
    void set();         // set state: write
    void await_resume(); // reset state: read
};

void auto_reset_event::await_resume(){
    auto fd = get_eventfd(state);
    if (is_signaled(state)) // already signaled
        consume_event(fd);

    // reset
    this->state = static_cast<uint64_t>(fd);
}

void consume_event(int64_t efd) {
    if (read(efd, &efd, sizeof(efd)) == -1)
        throw system_error{errno, system_category(), "read"};
}

```



Resume: Event Reset

```

void auto_reset_event::set() {
    if (is_signaled(this->state))
        return;

    auto fd = get_eventfd(this->state);
    notify_event(fd); // write
    this->state = emask | static_cast<uint64_t>(fd);
}

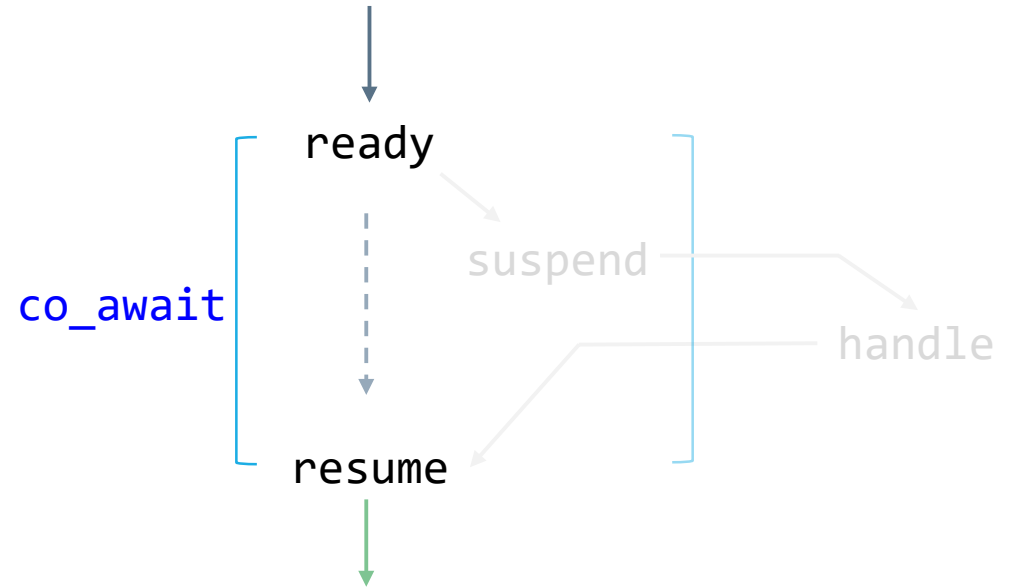
```

```

void auto_reset_event::await_resume() {
    auto fd = get_eventfd(state);
    if (is_signaled(state))
        consume_event(fd); // read

    // reset
    this->state = static_cast<uint64_t>(fd);
}

```



이미 Set(Write) 된 경우: read/Write 일치

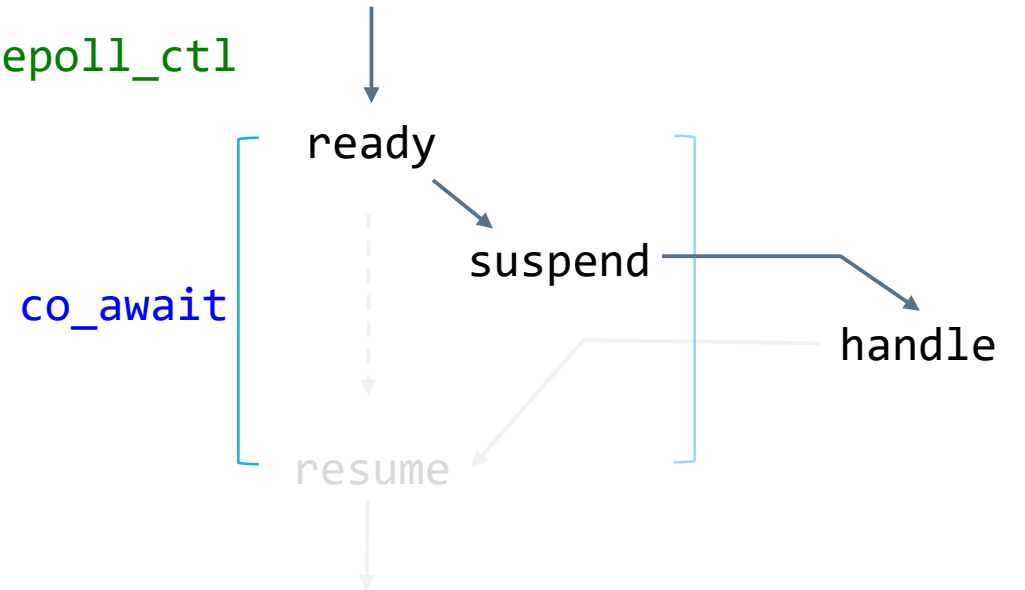
```
class auto_reset_event {
    uint64_t state;

public:
    void await_suspend(coroutine_handle<void>); // epoll_ctl
};
```

```
// signaled event list(global)
event_poll_t selist{};
```

```
void auto_reset_event::await_suspend(coroutine_handle<void> coro) {
    epoll_event req{};
    req.events = EPOLLET | EPOLLIN | EPOLLONESHOT;
    req.data.ptr = coro.address();

    // throws if `epoll_ctl` fails
    selist.try_add(get_eventfd(state), req);
}
```



Suspend: Registration

Q. 어째서 One-Time을 사용하는가?

System Object 연결이 해제되지 않는다면?

- 대표적으로 I/O Completion Port 같은 경우
- FD를 전달받았을 때 이 부분에 대한 추가적인 고려가 필요

사전조건^{Pre-condition}이 단순할 수록 Awaitable 타입의 구현이 쉬워진다

- 더 단순한 Ready/Suspend/Resume 코드
- 지금의 경우... `co_await`에 사용될때마다 Registration 처리

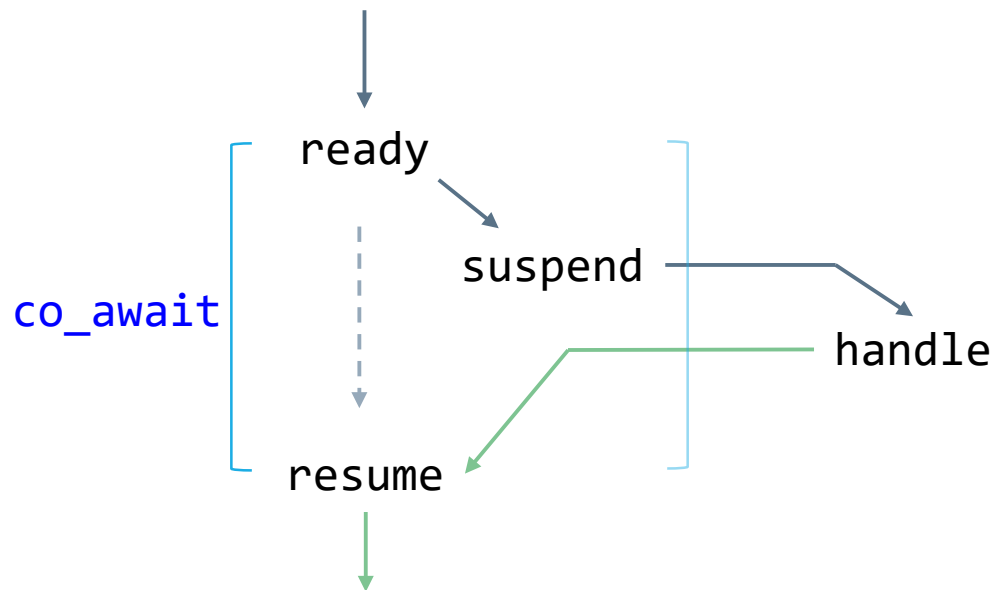
Q. await_suspend에서 예외를 던지는 이유?

Suspend 단계에서 오류가 발생한 경우

- Retry를 위해 재진입할 방법이 없음
- Awaitable 내부에 오류를 저장하고 Coroutine 외부에서 해결

실패할 수 있는 코드를 Ready로 옮긴다면?

- Resume 처리 과정이 복잡해질 가능성이 증가



```
class event_poll_t {  
    int epfd;  
    const size_t capacity;  
    unique_ptr<epoll_event[]> events;  
  
public:  
    event_poll_t();  
    ~event_poll_t();  
  
    void try_add(uint64_t fd, epoll_event& req);  
    auto wait(int timeout) -> generator<epoll_event>;  
};
```

좀 전에 봤던 epoll 개체의 Wrapper

```
class event_poll_t {  
    int epfd;  
    const size_t capacity;  
    unique_ptr<epoll_event[]> events;  
};
```

```
event_poll_t::event_poll_t(){  
    : epfd{-1},  
      capacity{2 * getpagesize() / sizeof(epoll_event)},  
      events{make_unique<epoll_event[]>(capacity)} {  
  
    epfd = epoll_create1(Epoll_Cloexec);  
    if (epfd < 0)  
        throw system_error(errno, system_category(), "epoll_create1");  
}  
  
event_poll_t::~~event_poll_t(){  
    close(epfd);  
}
```

```

class event_poll_t {
    int epfd;
    const size_t capacity;
    unique_ptr<epoll_event[]> events;

public:
    void try_add(uint64_t fd, epoll_event& req);
    auto wait(int timeout) -> generator<epoll_event>;
};

void event_poll_t::try_add(uint64_t _fd, epoll_event& req){ // << await_suspend
    int op = EPOLL_CTL_ADD, ec = 0;
TRY_OP:
    ec = epoll_ctl(epfd, op, _fd, &req);
    if (ec == 0)
        return;
    if (errno == EEXIST) { // already exists.
        op = EPOLL_CTL_MOD; // try with modification
        goto TRY_OP;
    }
    throw system_error{errno, system_category(), "epoll_ctl"};
}

```

Handle 등록: epoll_ctl


```

class event_poll_t {
    int epfd;
    const size_t capacity;
    unique_ptr<epoll_event[]> events;

public:
    void try_add(uint64_t fd, epoll_event& req);
    auto wait(int timeout) -> generator<epoll_event>;
};

auto event_poll_t::wait(int timeout) -> generator<epoll_event> { // >> signaled_event_tasks
    auto count = epoll_wait(epfd, events.get(), capacity, timeout);
    if (count == -1)
        throw system_error{errno, system_category(), "epoll_wait"};

    for (auto i = 0; i < count; ++i) {
        co_yield events[i];
    }
}

```

Handle 획득: epoll_wait >> generator

```

auto signaled_event_tasks() -> generator<coroutine_handle<void>> {
    coroutine_handle<void> task{};

    for (epoll_event e : selist.wait(0)) { // epoll_wait + timeout(0)
        task = coroutine_handle<void>::from_address(e.data.ptr);
        if (task.done())
            throw logic_error{"coroutine_handle<void> is already done state"};

        co_yield task;
    }
    co_return;
}

auto resume_event_waiting_coroutines() {
    // coroutine_handle<void>
    for (auto coro : signaled_event_tasks())
        coro.resume();
}

```

```

typedef union epoll_data {
    void* ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

```

generator<T> 를 사용한 Handle 전달

평가^{Evaluation}: eventfd + epoll

목적

- Event Notification을 `co_await`하고싶다!

구현결과

- `auto_reset_event` & `signaled_event_tasks()`

분석 포인트

- `coroutine_handle<void>`를 `user_data(== void*)`로 사용하면서 얻게되는 간결함
- 중단과 재개의 분리 >> **Awaitable + Handle 관리**의 분리

한계점?

- Resume이 무작위 순서로 수행: Q. 특정 Coroutine이 먼저 Resume되어야 한다면?
- API 특성상 Polling은 피할 수 없다!
- Event Object에 Write 한 시점^{Timepoint}과 Read 하는 시점의 격차^{Interval}가 커질 수 있다

Windows: Callback + Coroutine

Resume in Win32 Thread Pool

목적 Motivation

굳이 Resume을 수동으로 해줘야 하나?

- == 별다른 조치 없이 Coroutine을 Resume할 수는 없을까?
- Thread Pool Callback 을 사용해보자

```
auto work_on_thread_pool(latch& wg) -> forget {  
    co_await ptp_work{}; // just 1 line to use thread pool  
  
    // ... Do some work ...  
    wg.count_down();  
}
```

분석Analysis

Windows System API와 관련해서 확인해야하는 내용들

- Callback을 사용한 Resume
- Win32 Thread Pool
 - `#include <threadpoolapiset.h>`

Callback and C++ Coroutines

일반적으로 Callback 들은 Function + Argument 형태

- 이때 많은 경우 `void*`를 사용
- == `coroutine_handle<void>`에 굉장히 적합

예를 들어, `pthread_create()`, `start_routine` 를 사용한다면...

```
#include <pthread.h>
```

```
int pthread_create(pthread_t* thread, const pthread_attr_t* attr,  
                  void* (*start_routine)(void*), void* arg);
```

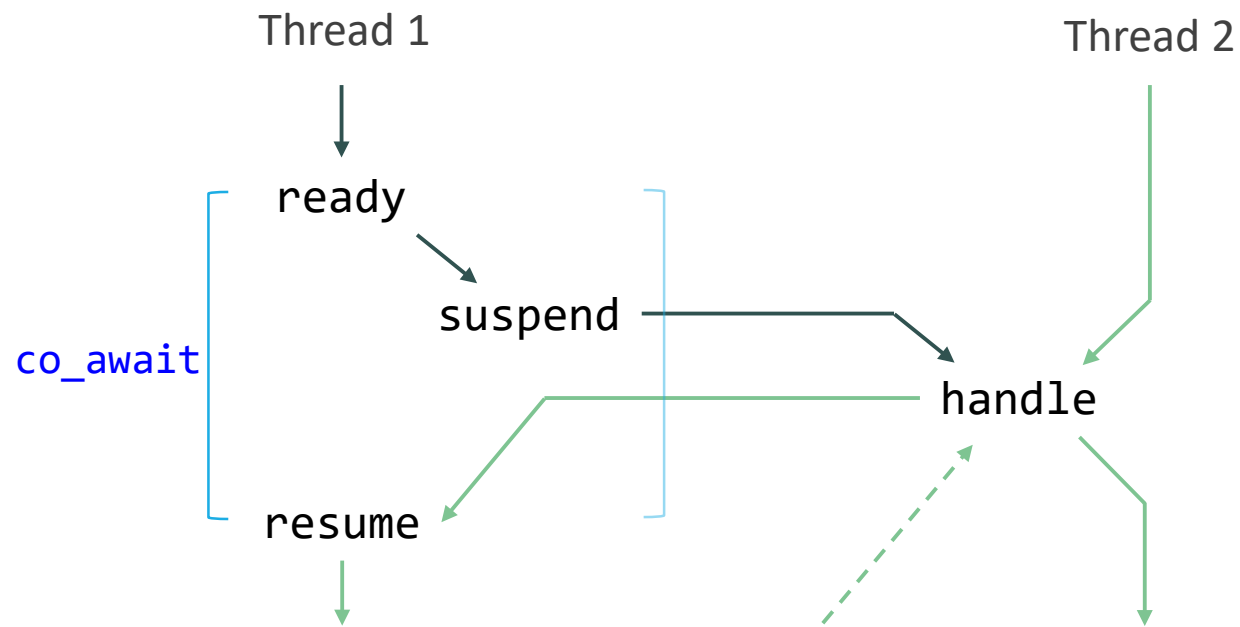
```
void* resumer(void* arg) {  
    if (auto coro = coroutine_handle<void>::from_address(arg))  
        coro.resume();  
    return arg;  
}
```

Callback and C++ Coroutines

하지만 Spawner Thread와 Callback Thread가 서로 다르다면?

- **Handle(Frame)에 대한 Data Race** 발생 가능성

동시성 문제는 Promise, Awaitable 설계에 영향을 줄 수 있음



Win32 Thread Pool

Win32 API에서는 이미 쉽게 Thread Pool / Thread Pool Callback을 사용할 수 있도록 지원

- docs.microsoft.com/en-us/windows/win32/procthread/using-the-thread-pool-functions

지원 기능

- Thread Pool Create/Close
- Resource Clean Up
- Timer: devblogs.microsoft.com/cppblog/coroutines-in-visual-studio-2015-update-1/
- **User Work Item (Task)**

```
auto work_on_thread_pool(latch& wg) -> forget {
    co_await ptp_work{}; // just 1 line to use thread pool

    // ... Do some work ...
    wg.count_down();
}

void fork_join(size_t num_worker) {
    latch group{num_worker};

    for (auto i = size_t{}; i < num_worker; ++i){
        work_on_thread_pool(group);
    }

    group.wait();
}
```

<https://en.cppreference.com/w/cpp/experimental/latch>

설계^{Design}: Look & Feel

```
class ptp_work final {
    // Callback for CreateThreadpoolWork
    static void __stdcall resume_on_thread_pool(PTP_CALLBACK_INSTANCE, PVOID,
                                                PTP_WORK);
    auto create_and_submit_work(coroutine_handle<void>) -> uint32_t;

public:
    bool await_ready() const;
    void await_resume();
    void await_suspend(coroutine_handle<void> coro);
};
```

Thread Pool에
Handle을 전달하기 위한 Awaitable

```

class ptp_work final {
    // Callback for CreateThreadpoolWork
    static void __stdcall resume_on_thread_pool(PTP_CALLBACK_INSTANCE, PVOID,
                                                PTP_WORK);

    auto create_and_submit_work(coroutine_handle<void>) -> uint32_t;

public:
    bool await_ready() const {
        return false;
    }
    void await_resume() {
        // nothing to do
    }
    void await_suspend(coroutine_handle<void> coro) {
        if (auto ec = create_and_submit_work(coro))
            throw system_error{ec, system_category(),
                               "create_and_submit_work"};
    }
};

```

□ □ □ □ □ □ □ □ Suspend

Thread Pool API로 □ □

ptp_work: Awaitable Functions 내부

```

class ptp_work final {
    // Callback for CreateThreadpoolWork
    static void __stdcall resume_on_thread_pool(PTP_CALLBACK_INSTANCE, PVOID,
                                                PTP_WORK);

    auto create_and_submit_work(coroutine_handle<void>) -> uint32_t;
    // ...
};

void ptp_work::resume_on_thread_pool(PTP_CALLBACK_INSTANCE, //
                                     PVOID ctx, PTP_WORK work) {
    if (auto coro = coroutine_handle<void>::from_address(ctx))
        if (coro.done() == false)
            coro.resume();

    ::CloseThreadpoolWork(work); // one-time work item
}

```

```

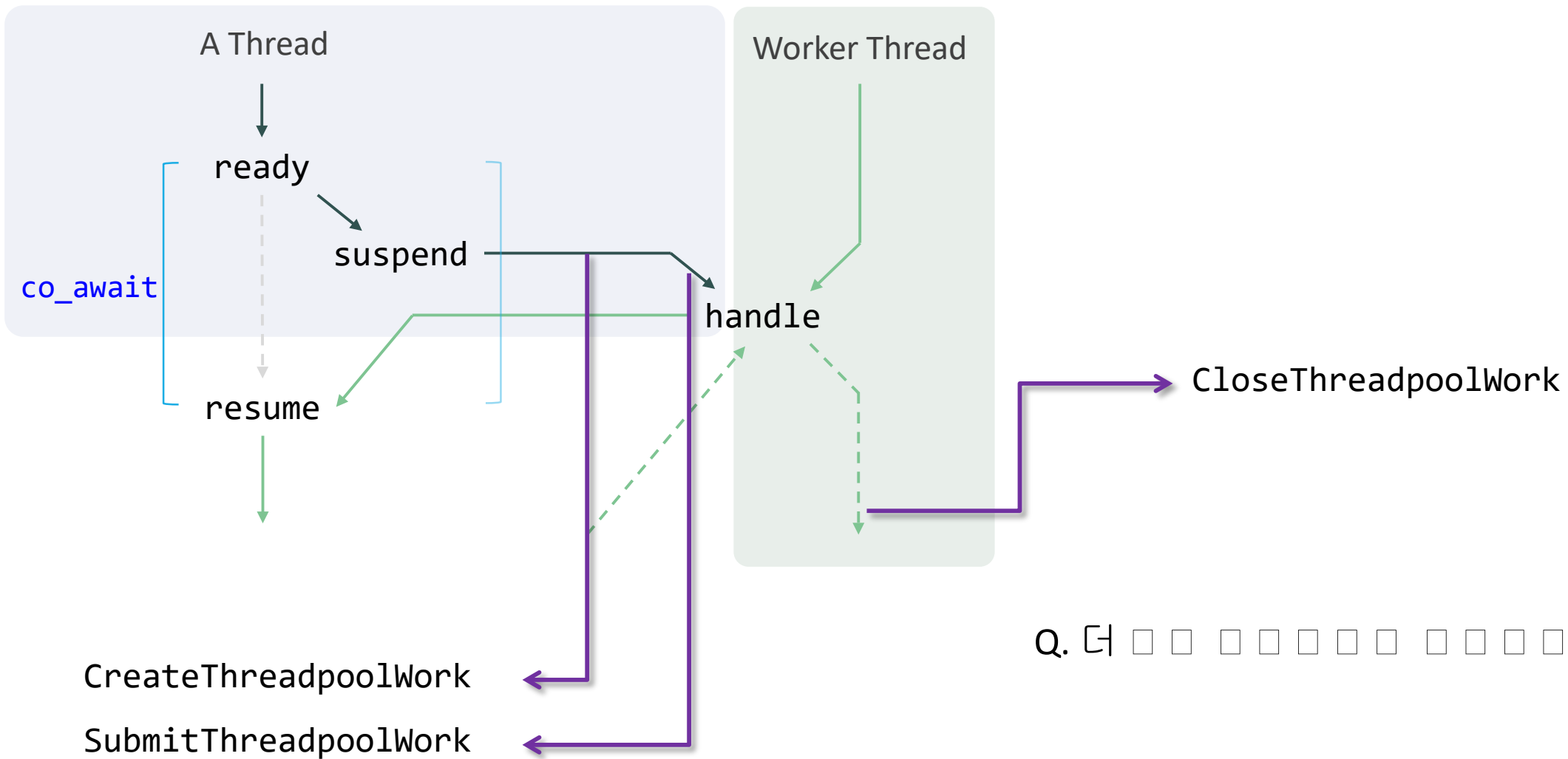
class ptp_work final {
    // Callback for CreateThreadpoolWork
    static void __stdcall resume_on_thread_pool(PTP_CALLBACK_INSTANCE, PVOID,
                                                PTP_WORK);

    auto create_and_submit_work(coroutine_handle<void>) -> uint32_t;
    // ...
};

auto ptp_work::create_and_submit_work(coroutine_handle<void> coro) -> uint32_t {
    // work := resume callback + handle
    auto work = ::CreateThreadpoolWork(resume_on_thread_pool, coro.address(), nullptr);
    if (work == nullptr)
        return GetLastError();

    ::SubmitThreadpoolWork(work); // >> resume_on_thread_pool
    return S_OK;
}

```



API 배치

```
auto work_on_thread_pool(latch& wg) -> forget {  
    // ...  
    co_await ptp_work{};  
    // ... Do some work ...  
    wg.count_down();  
}
```

A Thread

Worker Thread

```
void fork_join(size_t num_worker) {  
    latch group{num_worker};  
  
    for (auto i = size_t{}; i < num_worker; ++i){  
        work_on_thread_pool(group);  
    }  
  
    group.wait();  
}
```

A Thread

Code Coverage

평가^{Evaluation}: Callback + Coroutine

목적

- Callback을 Resume 하고싶다!

구현결과

- `ptp_work`: Thread Pool에 의한 Resume

분석 포인트

- 이미 Callback을 사용하는 경우 Coroutine을 쉽게 적용 가능
- 보다 명확한 전달력

한계점?

- 경우에 따라 동시성 이슈를 고려 (Ex. `latch`)

FreeBSD: Socket operation + kqueue

Awaitable Socket Operation

목적 Motivation

Linux API를 사용했던 경험을 Socket에 적용해본다면 어떨까?

- Eventfd --> Socket
- Epoll --> Kqueue

Boost ASIO보다 간결하게 작성할 수 있도록 한다면?

```
using io_buffer_reserved_t = array<std::byte, 3900>;

auto sender_coroutine(int64_t sd, io_work_t& work,
                      const sockaddr_in& remote) -> forget {
    // buffer in the coroutine frame
    io_buffer_reserved_t storage{};

    int64_t sz = co_await send_to(sd, remote, storage, work);
    auto ec = work.error();
}
```

분석Analysis

FreeBSD System API와 관련해서 확인해야하는 내용들

- kqueue 와 epoll의 차이점
- eventfd와 Socket의 차이점

kqueue

FreeBSD에서의 POSIX select/poll

Linux epoll과의 차이점?

- Event Filter 기반 (epoll: Trigger 기반)
- 하나의 함수에 더 복잡한 인자 타입 (epoll: `epoll_ctl`/`epoll_wait`로 분리되어 있음)

잠깐 살펴보자면 ...

```
#include <sys/epoll.h>
```

```
int epoll_create1(int flags);
```

Epoll Object 생성

```
int epoll_ctl(int epfd, int op,  
              int fd, struct epoll_event* event);
```

Epoll에 FD를 등록

```
struct epoll_event {  
    uint32_t events; // epoll events  
    epoll_data_t data; // user's data  
};
```

```
typedef union epoll_data {  
    void* ptr;  
    int fd;  
    uint32_t u32;  
    uint64_t u64;  
} epoll_data_t;
```

coroutine_handle<void> == void*

누적된 Event 확인

```
int epoll_wait(int epfd, struct epoll_event* events, int maxevents,  
               int timeout);
```

epoll 관련 함수들: 다시보기

```
#include <sys/event.h>
#include <sys/time.h>
#include <sys/types.h>
```

kqueue Object 생성

```
int kqueue();
```

```
struct kevent {
    uintptr_t ident; // identifier for this event
    int16_t filter; // filter for event
    uint16_t flags; // action flags for kqueue
    uint32_t fflags; // filter flag value
    int64_t data; // filter data value
    void* udata; // opaque user data
};
```

coroutine_handle<void> == void*

```
int kevent(int kq,
           const struct kevent* changelist, int nchanges, // Kqueue object 변경
           struct kevent* eventlist, int nevents, // Kqueue 누적 이벤트 획득
           const timespec* timeout);
```

kqueue 관련 함수들

Socket vs. eventfd

Socket 은 상당히 세밀한 제어가 필요

- Protocol에 따라서는 Scatter/Gather 처리 (eventfd: 고정 길이 Read/Write)
- Layer에 따른 Option 관리 (eventfd: 고려할 필요 없음)
- 복잡한 오류 원인과 처리 방법 (eventfd: 매우 한정적. Ex. Write Overflow)
- Kernel 내 버퍼 >> User 의 데이터 개체 (eventfd: 언제나 host endian `uint64_t`)
`byte stream >> struct + host endian 변환`
- Transport 보다 상위 Layer에서는 Encryption, Compression 등도 수행

Socket 을 Wrapping 하기 시작하면 eventfd 보다 훨씬 많은 정보량을 관리해야 한다!

설계^{Design}: Awaitable

Socket 대신 I/O Operation을 Awaitable로 사용

- `io_work_t` + 하위 타입들 Class Hierarchy
- 어떤 정보를 묶어서 Organize 처리할 것인가?
 - Address : Protocol에 따라서 필요
 - Error Code : 마지막 Operation의 `errno`
 - Buffer Range : `byte*` + `size_t` 형태로 (Scatter/Gather 포기)

Ready

- Socket의 Non-blocking 검사: Blocking Socket이면 Bypass

Suspend

- Global kqueue object에 등록

Resume

- 실제 Operation 처리

설계^{Design}: Handle

I/O Multiplex 하는 구조는 epoll과 같은 방법을 사용

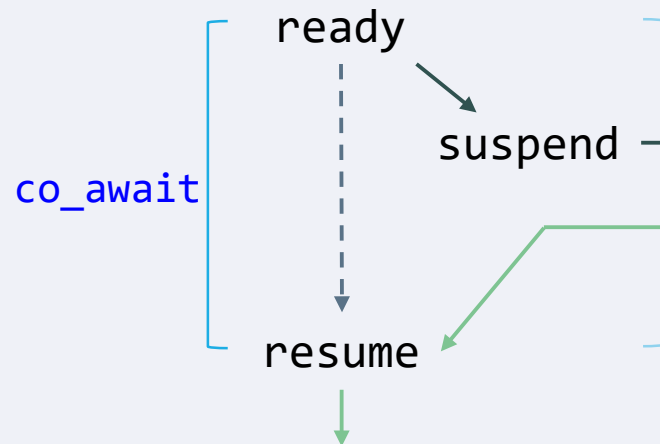
== Resume 가능한 Coroutine들을 일제히 반환

```
using io_task_t = coroutine_handle<void>;

auto wait_net_tasks(chrono::nanoseconds timeout) -> generator<io_task_t>;

void io_multiplex_subroutine() {
    for (auto task : wait_net_tasks(2ms)) // wait shortly
        task.resume();                  // and resume every availables
}
```

Awaitable



Handle

handle

wait_net_tasks

: kevent(event)

`io_work_t / io_send_to / io_rcv_from`

- `await_ready` : `fcntl`
- `await_suspend` : `kevent(change)`
- `await_resume` : `sendto/rcvfrom`

Epoll + `eventfd`와 마찬가지로
전역변수(Global object)를 사용

API 배치

```
using io_task_t = coroutine_handle<void>; // 1 I/O task == 1 coroutine function
using io_buffer_t = gsl::span<std::byte>; // == C++ 20 <span>
```

```
class io_work_t : public io_control_block {
```

```
public:
```

```
    io_task_t task{};
```

```
    io_buffer_t buffer{};
```

```
public:
```

```
    bool await_ready();
```

```
    uint32_t error();
```

```
};
```

Control block

- ???

Ready

- 하위 타입에서 사용
- Non-blocking Socket인지 검사

Error Check

- 가장 마지막 Error(uint32_t)를 저장

io_work_t: Awaitable을 위한 기본 타입

```

using io_task_t = coroutine_handle<void>;
using io_buffer_t = gsl::span<std::byte>;

class io_work_t : public io_control_block {
public:
    io_task_t task{};
    io_buffer_t buffer{};
};

struct io_control_block {
    uint64_t internal;
    uint64_t internal_high;
    union {
        struct {
            int32_t offset;
            int32_t offset_high;
        };
        void* ptr;
    };
    int64_t handle;
};

```

이유:

- Storage는 항상 필요하므로, 가능하다면 Portable하게!

역할:

- Before API : Argument 저장
- After API : Operation 결과를 저장

`io_control_block` == Win32 OVERLAPPED

```
class io_work_t : public io_control_block {  
public:  
    bool await_ready();  
    uint32_t error();  
};
```

```
bool io_work_t::await_ready() {  
    auto sd = this->handle;  
    if (fcntl(sd, F_GETFL, 0) & O_NONBLOCK)  
        return false;  
  
    return true;  
}
```

Non-blocking:

- **false**: 중단하면서 kqueue에 등록

Blocking:

- **true**: 중단 없이 바로 진행

```
uint32_t io_work_t::error() {  
    return static_cast<uint32_t>(this->internal);  
}
```

Error Check:

- 내부에 Error Code를 저장

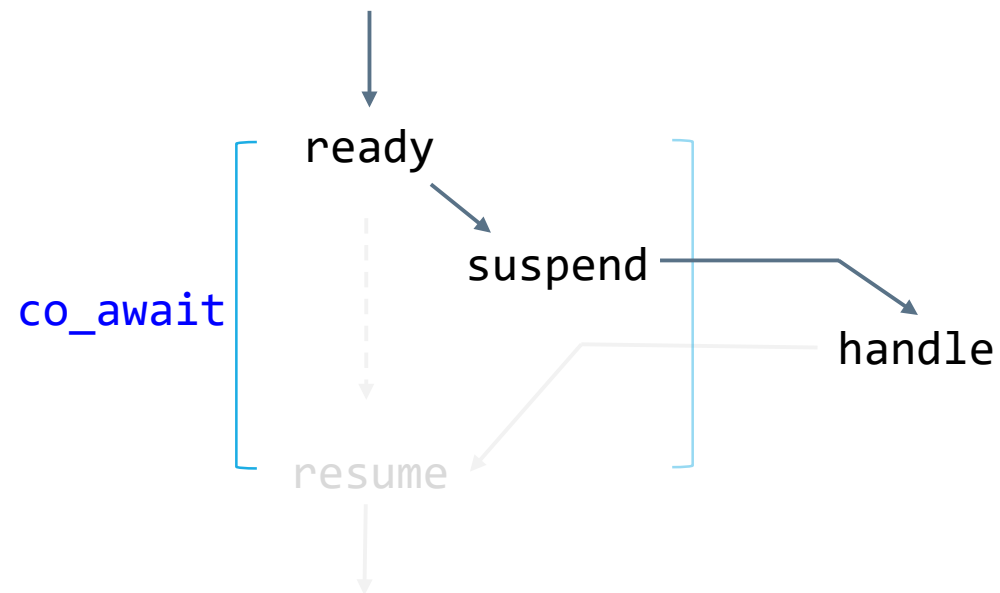
```
class io_work_t : public io_control_block {
public:
    bool await_ready();
    uint32_t error();
};

class io_send_to final : public io_work_t {
public:
    void await_suspend(io_task_t t);
    int64_t await_resume();
};
static_assert(sizeof(io_send_to) == sizeof(io_work_t));
```

Awaitable Operation Type:
io_send_to

```
class io_work_t : public io_control_block {
public:
    bool await_ready();
    uint32_t error();
};
```

```
class io_send_to final : public io_work_t {
public:
    void await_suspend(io_task_t t);
    int64_t await_resume();
};
static_assert(sizeof(io_send_to) == sizeof(io_work_t));
```



Suspend
◦ Kqueue 등록

Suspend 명세:
io_send_to


```
class io_work_t : public io_control_block {
public:
    bool await_ready();
    uint32_t error();
};
```

```
class io_send_to final : public io_work_t {
public:
    void await_suspend(io_task_t coro) {
        task = coro;
```

```
        kevent64_s req{};
        req.ident = this->handle; // socket
        req.filter = EVFILT_WRITE; // one-shot, WRITE
        req.flags = EV_ADD | EV_ENABLE | EV_ONESHOT;
        req.fflags = 0;
        req.data = 0;
        req.udata = reinterpret_cast<uint64_t>(
            static_cast<io_work_t*>(this));
```

```
        kq.change(req); // kevent(change)
    }
```

I/O Operation Type

- Instance는 Kqueue에 전달할 인자들의 집합

Suspend

- Kqueue 등록
- Epoll에서처럼 One-time

Suspend 구현:
io_send_to

```

class io_work_t : public io_control_block {
public:
    bool await_ready();
    uint32_t error();
};

class io_send_to final : public io_work_t {
public:
    void await_suspend(io_task_t coro) {
        task = coro;

        kevent64_s req{};
        req.ident = this->handle; // socket
        req.filter = EVFILT_WRITE; // one-shot, WRITE
        req.flags = EV_ADD | EV_ENABLE | EV_ONESHOT;
        req.fflags = 0;
        req.data = 0;
        req.udata = reinterpret_cast<uint64_t>(
            static_cast<io_work_t*>(this));

        kq.change(req); // kevent(change)
    }
}

```

Suspend 구현:
io_send_to

```
class io_work_t : public io_control_block {
public:
    bool await_ready();
    uint32_t error();
};

class io_recv_from final : public io_work_t {
public:
    void await_suspend(io_task_t coro) {
        task = coro;

        kevent64_s req{};
        req.ident = this->handle; // socket
        req.filter = EVFILT_READ; // one-shot, READ
        req.flags = EV_ADD | EV_ENABLE | EV_ONESHOT;
        req.fflags = 0;
        req.data = 0;
        req.udata = reinterpret_cast<uint64_t>(
            static_cast<io_work_t*>(this));

        kq.change(req); // kevent(change)
    }
};
```

Suspend 구현:
io_recv_from

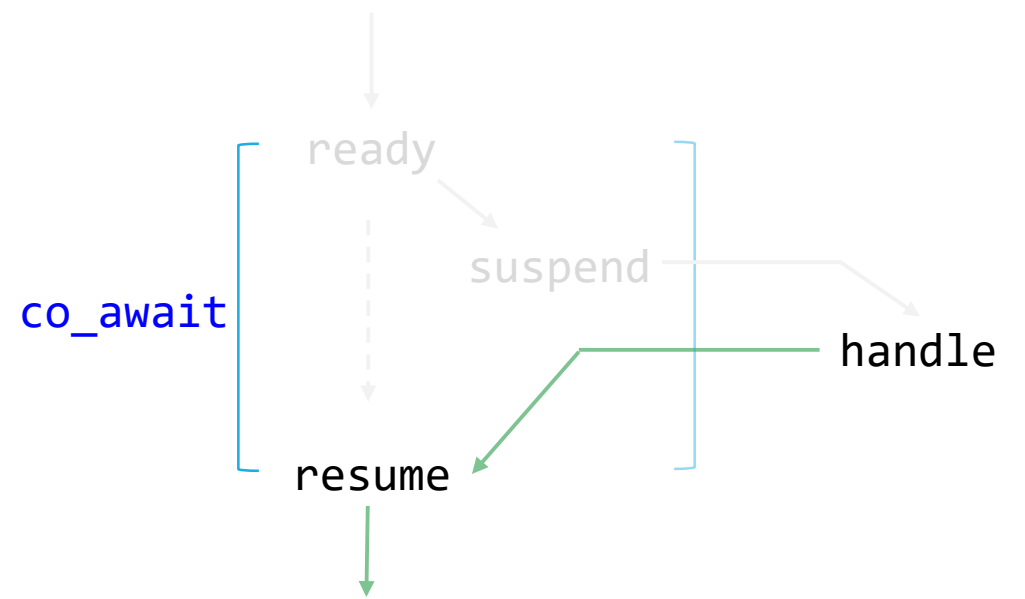
```

class io_work_t : public io_control_block {
public:
    bool await_ready();
    uint32_t error();
};

class io_send_to final : public io_work_t {
public:
    void await_suspend(io_task_t t);
    int64_t await_resume();
};

static_assert(sizeof(io_send_to) == sizeof(io_work_t));

```



Resume

- sendto 함수의 결과를 그대로 전달
- Error 확인

Resume 명세:
io_send_to

```

int64_t io_send_to::await_resume() {
    auto addr = reinterpret_cast<sockaddr*>(this->ptr);
    auto addrlen = static_cast<socklen_t>(this->internal_high);

    // operation
    auto sz = sendto(this->handle,           // socket
                    this->buffer.data(),    // gsl::span
                    this->buffer.size_bytes(), //
                    0, addr, addrlen);      // flag, remote

    // save error code
    this->internal = sz < 0 ? errno : 0;
    return sz;
}

```

The diagram illustrates the flow of data in the `await_resume()` function. A purple line originates from the `sendto` function call, specifically from the `sz` variable. It branches into two paths: one that goes directly to the `return sz;` statement, and another that goes to the `this->internal = sz < 0 ? errno : 0;` line. From there, the line continues to the `return sz;` statement, indicating that the error code is saved before returning the result.

Resume

- `sendto` 함수의 결과를 그대로 전달
- Error 확인

```

uint32_t io_work_t::error() {
    return static_cast<uint32_t>(this->internal);
}

```

Resume 구현:
io_send_to

```

int64_t io_recv_from::await_resume() {
    auto addr = reinterpret_cast<sockaddr*>(this->ptr);
    auto addrlen = static_cast<socklen_t>(this->internal_high);

    // operation
    auto sz = recvfrom(this->handle,           // socket
                       this->buffer.data(),    // gsl::span
                       this->buffer.size_bytes(), //
                       0, addr, &addrlen);     // flag, remote

    // save error code
    this->internal = sz < 0 ? errno : 0;
    return sz;
}

```

Resume 구현:
io_recv_from

```
class io_work_t : public io_control_block {
public:
    bool await_ready();
    uint32_t error();
};

class io_recv_from final : public io_work_t {
public:
    void await_suspend(io_task_t t);
    int64_t await_resume();
};
static_assert(sizeof(io_recv_from) == sizeof(io_work_t));
```

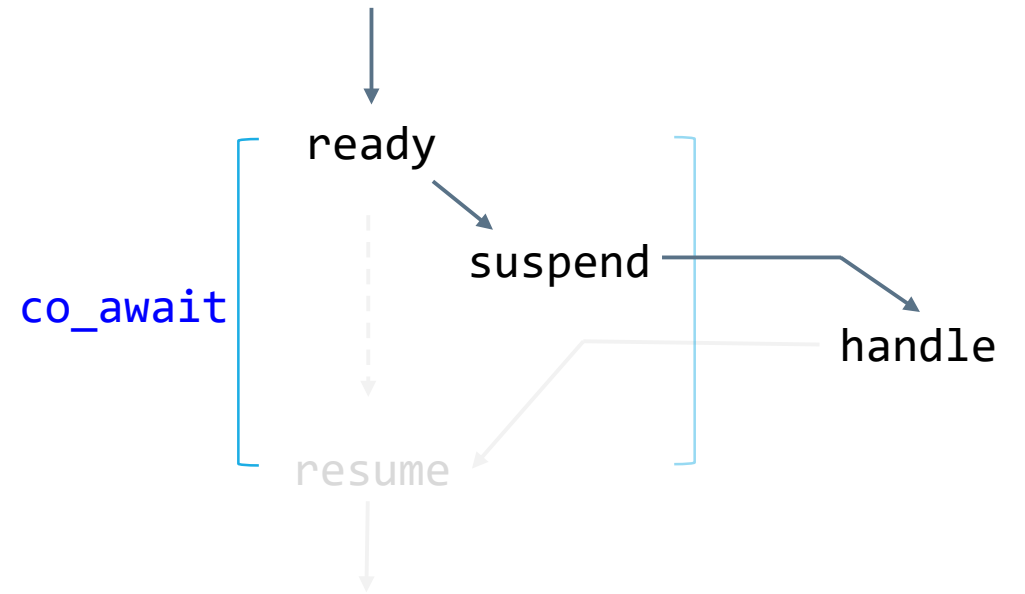
Awaitable Operation Type:
io_recv_from

```

class io_work_t : public io_control_block {
public:
    bool await_ready();
    uint32_t error();
};

class io_recv_from final : public io_work_t {
public:
    void await_suspend(io_task_t t);
    int64_t await_resume();
};
static_assert(sizeof(io_recv_from) == sizeof(io_work_t));

```



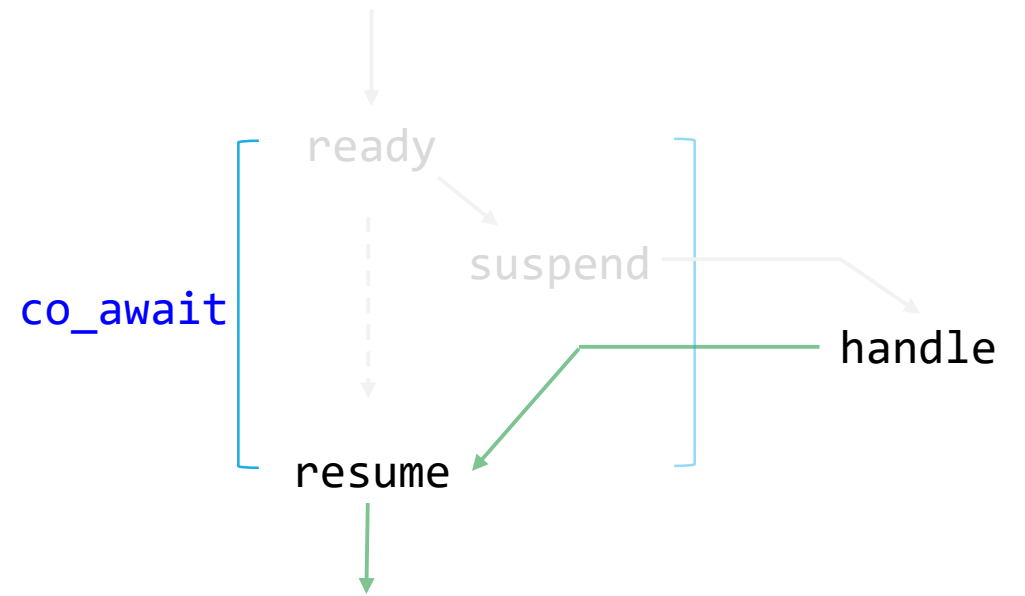
Suspend 명세:
io_recv_from


```

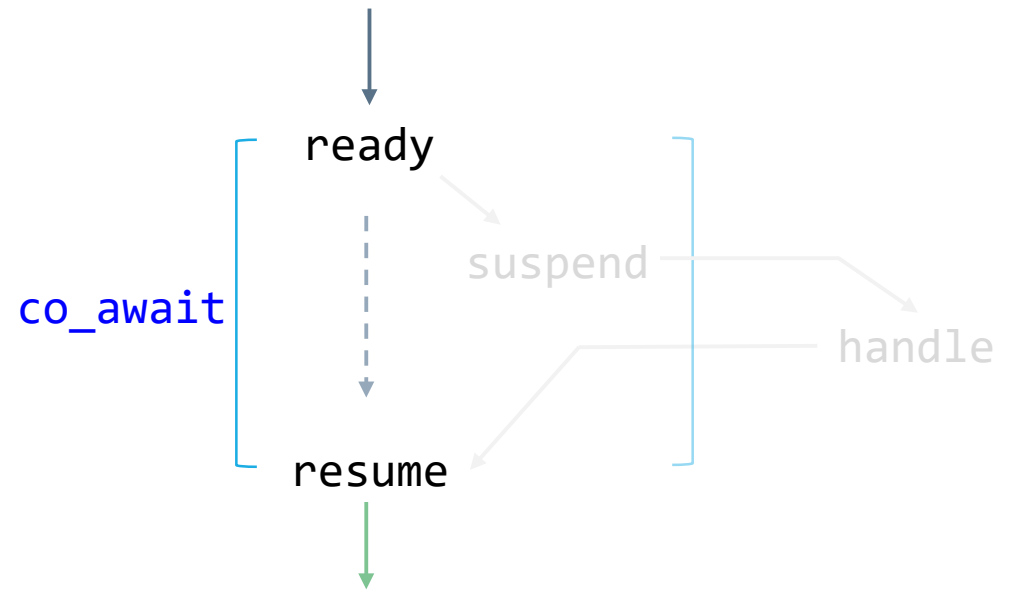
class io_work_t : public io_control_block {
public:
    bool await_ready();
    uint32_t error();
};

class io_recv_from final : public io_work_t {
public:
    void await_suspend(io_task_t t);
    int64_t await_resume();
};
static_assert(sizeof(io_recv_from) == sizeof(io_work_t));

```



Resume 명세:
io_recv_from



```
auto recv_datagram(uint64_t sd, io_work_t& work) -> forget {  
    sockaddr_in remote{};  
    io_buffer_reserved_t storage{};  
  
    int64_t rsz = co_await recv_from(sd, remote, storage, work);  
    if (auto ec = work.error()) {  
        // handle the error ...  
    }  
    // ...  
}
```

Use-case 재확인Remind

```

auto recv_from(uint64_t sd, sockaddr_in& remote, io_buffer_t buffer,
               io_work_t& work) -> io_recv_from& {
    work.handle = sd;
    work.ptr = addressof(remote);
    work.internal_high = sizeof(sockaddr_in);
    work.buffer = buffer;
    // convert 'io_work_t' to 'io_recv_from'
    return *reinterpret_cast<io_recv_from*>(addressof(work));
}

```

```

auto recv_datagram(uint64_t sd, io_work_t& work) -> forget {
    sockaddr_in remote{};
    io_buffer_reserved_t storage{};

    int64_t rsz = co_await recv_from(sd, remote, storage, work);
    if (auto ec = work.error()) {
        // handle the error ...
    }
    // ...
}

```

인자들로 Awaitable Type 만들기

```

auto recv_from(uint64_t sd, sockaddr_in& remote, io_buffer_t buffer,
               io_work_t& work) -> io_recv_from& {
    work.handle = sd;
    work.ptr = addressof(remote);
    work.internal_high = sizeof(sockaddr_in);
    work.buffer = buffer;
    // convert 'io_work_t' to 'io_recv_from'
    return *reinterpret_cast<io_recv_from*>(addressof(work));
}

```

```

auto recv_datagram(uint64_t sd, io_work_t& work) -> forget {
    sockaddr_in remote{};
    io_buffer_reserved_t storage{};

    int64_t rsz = co_await recv_from(sd, remote, storage, work);
    if (auto ec = work.error()) {
        // handle the error ...
    }
    // ...
}

```

단순한 `reinterpret_cast`!

```
auto recv_from(uint64_t sd, sockaddr_in& remote, io_buffer_t buffer,  
               io_work_t& work) -> io_recv_from&;  
  
auto send_to(uint64_t sd, const sockaddr_in& remote, io_buffer_t buf,  
             io_work_t& work) -> io_send_to&;  
  
auto recv_stream(uint64_t sd, io_buffer_t buffer, uint32_t flag,  
                 io_work_t& work) -> io_recv&;  
  
auto send_stream(uint64_t sd, io_buffer_t buffer, uint32_t flag,  
                 io_work_t& work) -> io_send&;
```

Q. 내부가 상상이 되시나요?

```

auto wait_net_tasks(choro::nanoseconds timeout) -> generator<io_task_t> {
    timespec ts{};
    const auto sec = duration_cast<seconds>(timeout);
    ts.tv_sec = sec.count();
    ts.tv_nsec = (timeout - sec).count();

    for (kevent64_s& ev : kq.wait(ts)) { // kevent(event)
        io_work_t* work = reinterpret_cast<io_work_t*>(ev.udata);
        // ... point to information
        //      from 'kevent64_s' to 'io_work_t' ...
        co_yield work->task;
    }
}

```

Kqueue 에서 Event 획득

```

auto wait_net_tasks(choro::nanoseconds timeout) -> generator<io_task_t> {
    timespec ts{};
    const auto sec = duration_cast<seconds>(timeout);
    ts.tv_sec = sec.count();
    ts.tv_nsec = (timeout - sec).count();

    for (kevent64_s& ev : kq.wait(ts)) { // kevent(event)
        io_work_t* work = reinterpret_cast<io_work_t*>(ev.udata);
        // ... point to information
        //      from 'kevent64_s' to 'io_work_t' ...
        co_yield work->task;
    }
}

void io_send_to::await_suspend(io_task_t coro) {
    // ...
    kq.change(req); // kevent(change)
}

```

The diagram consists of two purple lines. The first line starts at the `kq.change(req);` line in the `io_send_to::await_suspend` function and points down to the `for` loop in the `wait_net_tasks` function. The second line starts at the `co_yield work->task;` line inside the `for` loop and points back up to the `for` loop header, indicating a loop iteration.

Kqueue 는 어떻게 처리하고 있을까?

```
class kernel_queue_t final {  
    int kqfd;  
    const size_t capacity;  
    unique_ptr<kevent64_s[]> events;  
  
public:  
    kernel_queue_t();  
    ~kernel_queue_t();  
  
    void change(kevent64_s& req);  
    auto wait(const timespec& ts) -> generator<kevent64_s>;  
};
```

Epoll Wrapper와 거의 동일!


```

class kernel_queue_t final {
    int kqfd;
    const size_t capacity;
    unique_ptr<kevent64_s[]> events;

public:
    void change(kevent64_s& req);
    auto wait(const timespec& ts) -> generator<kevent64_s>;
};

void kernel_queue_t::change(kevent64_s& req) {

    auto ec = kevent64(kqfd, &req, 1, // change list (only 1)
                      nullptr, 0, 0, nullptr); // event list (empty)
    if (ec == -1)
        throw system_error{errno, system_category(),
                           "kevent64(change)"};
}

```

Registration: Kevent(change)

```

class kernel_queue_t final {
    int kqfd;
    const size_t capacity;
    unique_ptr<kevent64_s[]> events;

public:
    void change(kevent64_s& req);
    auto wait(const timespec& ts) -> generator<kevent64_s>;
};

auto kernel_queue_t::wait(const timespec& ts) -> generator<kevent64_s> {

    auto count = kevent64(kqfd, nullptr, 0, // change list (empty)
                          events.get(), capacity, 0, &ts); // event list (user buffer)
    if (count == -1)
        throw system_error{errno, system_category(),
                           "kevent64(event)"};

    for (auto i = 0; i < count; ++i) {
        co_yield events[i]; // deliver events ...
    }
}

```

Wait: Kevent(event)

```

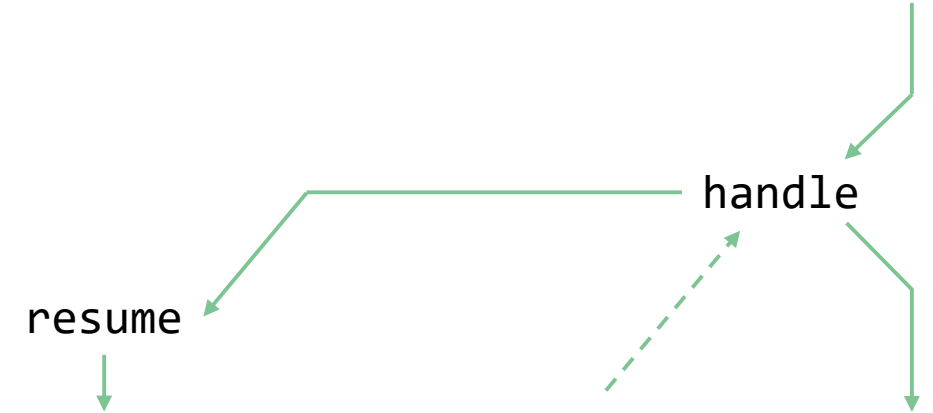
auto wait_net_tasks(choro::nanoseconds timeout) -> generator<io_task_t> {
    timespec ts = // timeout
    // ...
    for (kevent64_s& ev : kq.wait(ts)) { // kevent(event)
        io_work_t* work = // ev.udata
        // ...
        co_yield work->task;
    }
}

```

```

void io_multiplex_subroutine() {
    // wait shortly and resume every availables
    for (auto task : wait_net_tasks(2ms))
        task.resume();
}

```



Kqueue에서 Resume 까지

평가^{Evaluation}: Socket Operation + kqueue

이번 Section을 시작할때 떠올렸던 코드를 리뷰해봅시다

```
auto sender_coroutine(int64_t sd, io_work_t& work,
                      const sockaddr_in& remote) -> forget {
    io_buffer_reserved_t storage{};

    int64_t sz = co_await send_to(sd, remote, storage, work);
    auto ec = work.error();
}

void io_multiplex_subroutine() {
    // wait shortly and resume every availables
    for (auto task : wait_net_tasks(2ms))
        task.resume();
}
```

평가^{Evaluation}: Socket Operation + kqueue

목적

- `co_await` Socket Operation

구현결과

- `io_work_t >> io_send_to / io_recv_from`
- `wait_net_tasks()`

분석 포인트

- Operation을 Awaitable Type으로 사용: Operation에 담긴 의미가 내부 Argument들을 결정
- Polling은 여전히 필요: Q. 만약 Windows 였다면?

한계점?

- Operation이 다양할수록 구현코드가 더 많이 필요할 가능성

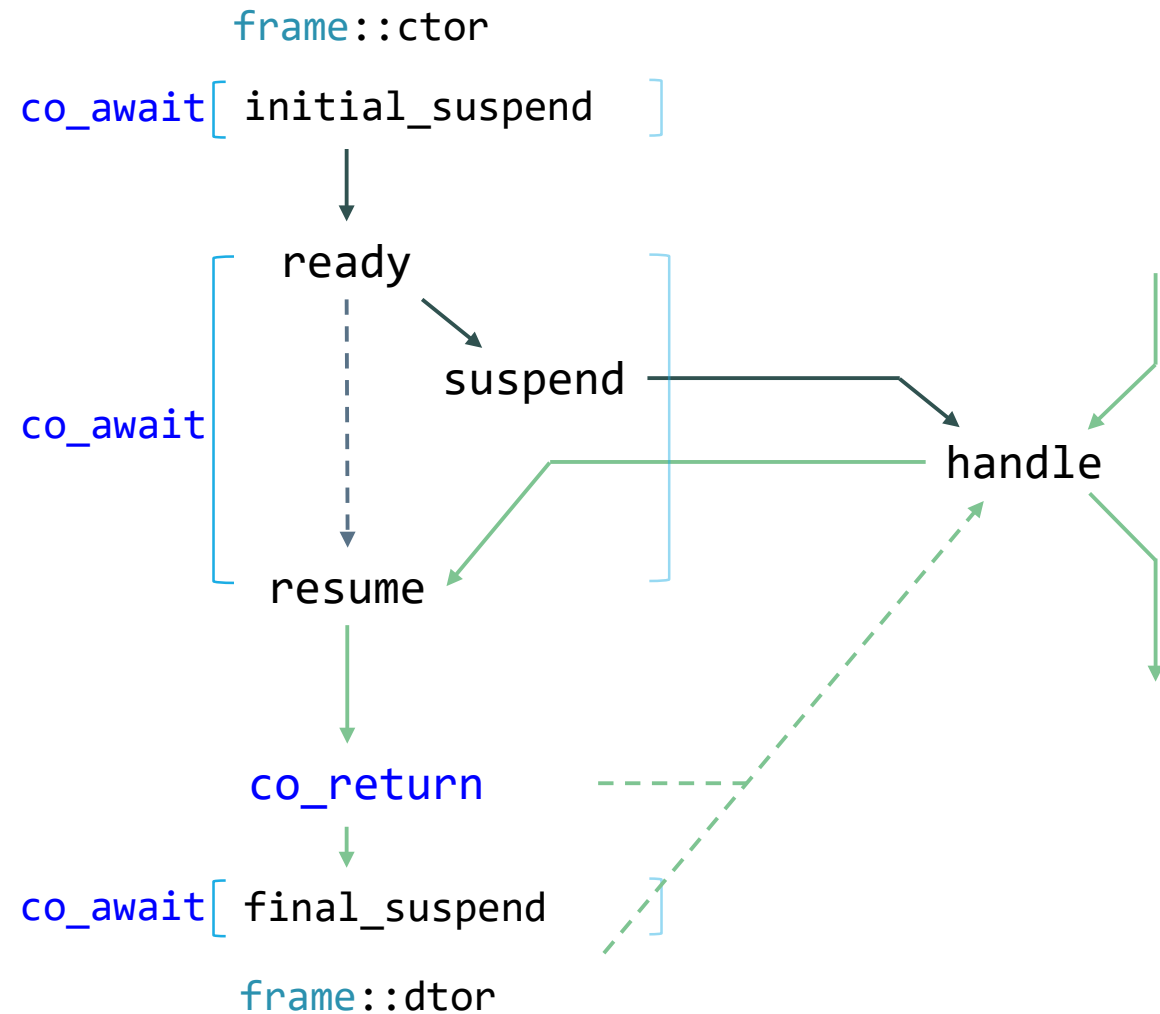
Conclusion

정리해봅시다!

발표에서 다룬 것들: C++ Coroutines

Coroutine 코드를 구성하는 부분

- Awaitable
- Promise
- Handle



발표에서 다룬 것들: System API

System API + Coroutine

- Linux: `auto_reset_event` + `signaled_event_tasks()`
- Windows: `ptp_work` (Win32 Thread Pool)
- FreeBSD: `io_send_to` / `io_recv_from` + `wait_net_tasks()`

코드를 분석해보면서 생각한 것들

분석 포인트

- `coroutine_handle<void>`를 `user_data(== void*)`로 사용하면서 얻게되는 간결함
- 중단과 재개의 분리: **Handle 관리 / Awaitable 사용**
- Operation을 Awaitable Type으로 사용
- 이미 Callback을 사용하는 경우 Coroutine을 쉽게 적용 가능

한계점

- API 특성상 Polling이 필요한 경우
 - Resume이 무작위 순서로 수행될 수 있음
- Operation이 다양할수록 구현코드가 더 많이 필요할 가능성
- 경우에 따라 동시성 이슈를 고려 (Ex. `latch`)

감사합니다! Q & A?

개발자 프로필 / 발표 저장소

- github.com/luncliff
- github.com/luncliff/coroutine

관련 질문 + 발표자료 오류 제보:

- luncliff@gmail.com