Coroutines and Structured Concurrency in Practice

Dmitry Prokoptsev
Stackless coroutines are there in C++ for 3 years

Lots of talks at conferences

Still rarely used in production
Challenges we faced

A lot of existing code
   mostly callback-based

A custom-built I/O event loop
   predates Asio by a decade
   fine-tuned for specific use cases

Somewhat conservative audience
   real benefit needs to be demonstrated to justify switching
Callbacks vs coroutines

// read length
auto len = make_unique<int32_t>();
async_read(socket,
    buffer(len.get(), sizeof(int32_t)),
    [len = move(len)](auto ec, size_t len1) {
        if (!ec && len1 == sizeof(int32_t)) {
            // read data
            auto buf = make_unique<char[]>(*len);
            async_read(socket,
                buffer(buf.get(), *len),
                [buf = move(buf), len = *len] (auto ec, size_t len2) {
                    if (!ec && len2 == len) {
                        // ...handle data...
                    } else {
                        // ...handle error...
                    }
                });
        } else {
            // ...handle error...
        }
    });

// read length
int32_t len;
co_await async_read(s,
    buffer(&len, sizeof(len)),
    use_awaitable);

// read data
auto buf = make_unique<char[]>(len);
co_await async_read(s,
    buffer(buf.get(), len),
    use_awaitable);

// ...handle data...

Local variable
Manual lifetime management
Callbacks vs coroutines

```
// read length
auto len = make_unique<int32_t>();
async_read(socket,
    buffer(len.get(), sizeof(int32_t)),
    [len = move(len)](auto ec, size_t len1){
        if (!ec && len1 == sizeof(int32_t)) {
            // read data
            auto buf = make_unique<char[]>(*len);
async_read(socket,
                buffer(buf.get(), *len),
                [buf = move(buf), len = *len]
                (auto ec, size_t len2) {
                    if (!ec && len2 == len) {
                        // ...handle data...
                    } else { // ...handle error...
                        }
                });
        } else { // ...handle error...
    }
});
```
Callbacks vs coroutines

// read length
auto len = make_unique<int32_t>();
async_read(socket,
    buffer(len.get(), sizeof(int32_t)),
    [len = move(len)](auto ec, size_t len1){
        if (!ec && len1 == sizeof(int32_t)) {
            // read data
            auto buf = make_unique<char[]>(*len);
            async_read(socket,
                buffer(buf.get(), *len),
                [buf = move(buf), len = *len]
                (auto ec, size_t len2) {
                    if (!ec && len2 == len) {
                        // ...handle data...
                    } else {
                        // ...handle error...
                    }
                });
        } else {
            // ...handle error...
        }
    });

// read length
int32_t len;
co_await async_read(s,
    buffer(&len, sizeof(len)),
    use_awaitable);

// read data
auto buf = make_unique<char[]>(len);
co_await async_read(s,
    buffer(buf.get(), len),
    use_awaitable);

// ...handle data...

Errors raised from here as C++ exceptions

Manual error handling
Coroutines may simplify things

• Easier reasoning

• Easier object lifetime management

• Easier error propagation

But we need some structure
Structured concurrency

And why we care
A typical async framework

class Task { ... }
    represents a unit of background work

Task::join()
    explicit call to suspend the current task
    until another task completes, and returns its result
    propagates uncaught exceptions

Task::detach()
    allows the task to run alongside the rest of the program
Detached tasks considered harmful

No way to figure out task lifetime
=> no automatic object lifetime management

```cpp
// don't do this
void bad(tcp::socket& s) {
    std::array<char> buf(1024);
    asio::co_spawn(
        ex,
        [&s, &buf]() -> asio::awaitable {
            co_await s.read_some(s,
                asio::buffer(buf),
                asio::use_awaitable);
        },
        asio::detached);
}
```
Detached tasks considered harmful

// don’t do this either
void slightly_better(tcp::socket& s) {
    auto buf = make_shared<char[]>(1024);
    asio::co_spawn(
        ex,
        [&](s, buf) -> asio::awaitable {
            co_await s.read_some(s,
                asio::buffer(buf.get(), 1024),
                asio::use_awaitable);
        },
        asio::detached);
}
Detached tasks considered harmful

// and also don’t do this
void slightly_better(tcp::socket& s) {
    auto buf = make_shared<char[]>(1024);
    asio::co_spawn(ex,
                   [&s, buf]()->asio::awaitable {
                     try {
                       co_await s.read_some(s,
                                            asio::buffer(buf.get(), 1024),
                                            asio::use_awaitable);
                     } catch (std::exception& e) {
                       // ...um?..
                     }
                   },
                   asio::detached);
}
Can we fix things by just removing \texttt{detach()}? 

Each task must be \texttt{join()}ed at some point

\texttt{join()} is an explicit call, we cannot \textit{force} users to always call it

Also join in destructor?

Error propagation still problematic

Destructors are not supposed to throw

What if task destructor \textit{is never reached}?
What if task destructor is never reached?

```cpp
void reportLiveness();
Task serve(tcp::socket s);

Task mainTask() {
    Task liveness = [](()) -> Task {
        for (;;) {
            reportLiveness();
            co_await sleepFor(1_s);
        }
    }();

    for (;;) {
        tcp::socket = co_await accept();
        co_await serve(std::move(s));
    }
}

// implicit liveness::~Task() call
```

If this throws, the exception gets reraised here.
What if task destructor is never reached?

```cpp
void reportLiveness();
Task serve(tcp::socket s);

Task mainTask() {
    Task liveness = []() -> Task {
        for (; ;) {
            reportLiveness();
            co_await sleepFor(1_s);
        }
    }();

    for (; ;) {
        tcp::socket = co_await accept();
        co_await serve(std::move(s));
    }

    std::unreachable();
    // implicit liveness.~Task() call
}
```

If this throws we’ll never find out
What if task destructor is never reached?

```cpp
void reportLiveness();
Task serve(tcp::socket s);

Task mainTask() {
    Task liveness = []() -> Task {
        for (;;) {
            reportLiveness();
            co_await sleepFor(1_s);
        }
    }();

    for (;;) {
        tcp::socket = co_await accept();
        co_await serve(std::move(s));
    }

    std::unreachable();
    // implicit liveness::~Task() call
}
```

Executes in background
The ground rule:
No such thing as background execution

A task can only run when it’s being awaited by another task
A task can only run when it’s being awaited by another task

That awaiting task is a caller

Once the caller resumes, the callee is done any resources it may have used can be freed

Any unhandled exception will get re-raised

https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/
Sketching an API

```cpp
Task<void> greet() {
    cout << "going to greet "
    << "the world\n";
    co_await sleep_for(1s);
    cout << "Hello world!\n";
}

Task<void> greetTwice() {
    cout << "spawning tasks\n";
    auto task1 = greet();
    auto task2 = greet();
    cout << "awaiting tasks\n";
    co_await task1;
    co_await task2;
}
```

spawning tasks
awaiting tasks
going to greet the world
  <1 second pause>
Hello world!
going to greet the world
  <another 1 second pause>
Hello world!
Running things concurrently

```cpp
Task<void> greet() {
    cout << "going to greet "
        << "the world\n";
    co_await sleep_for(1s);
    cout << "Hello world!\n";
}

Task<void> greetTwice() {
    co_await allOf(greet(),
                   greet());
}
```

- going to greet the world
- going to greet the world
- `<1 second pause>`
- Hello world!
- Hello world!
allOf() combiner

• Upon `co_await`, starts all children

• Once all children complete, returns `std::tuple<>` of their results

• If any child raises an exception, cancels anything still running and re-raises
anyOf() combiner

• Upon `co_await`, starts all children

• Once any of them completes, *cancels* the others

• Once cancellation completes, returns `std::variant<>` of their results

• If any child raises an exception, cancels anything still running and re-raises
anyOf() use cases

Attaching a timeout to a long-running operation

Task<
  void
longRunning();
anyOf() use cases

Attaching a timeout to a long-running operation

```
Task<void> longRunning(std::chrono::milliseconds timeout);
```

https://vorphic.org/blog/timeouts-and-cancellation-for-humans/
anyOf() use cases

Attaching a timeout to a long-running operation

```csharp
Task<
    void
>
longRunning();

Task<
    void
> bounded() {
    co_await anyOf(
        longRunning(),
        sleepFor(200ms));
}
```
anyOf() use cases

Making an operation externally cancellable

```cpp
Task<void> longRunning();
Event* evt = nullptr;
Task<void> cancellable() {
  Event e;
  evt = &e;

  co_await anyOf(longRunning(), e);
}

void cancel() { evt->trigger(); }
```
anyOf() use cases: clean shutdown

class Server {
public:
    Task<void> serve(); // runs forever
};

int main() {
    Server srv;
    run(anyOf(srv.serve(),
             signalReceived(\{SIGTERM, SIGINT\})));
}
anyOf() use cases: racing

Task<ip::addr> resolveOn(const string& name, const string& dnsServer);

Task<ip::addr> resolve(const string& name) {
    auto v = co_await anyOf(
        resolveOn(name, "8.8.8.8"),
        [&](){ -> Task<ip::addr> {
            co_await sleepFor(100ms);
            co_return co_await resolveOn(name, "1.1.1.1");
        }});

    co_return visit(identity{}, v);
}
Structured concurrency

Tasks naturally form a "call tree"
• coroutines
• leaf awaitables (sleeping, I/O, etc)
• combiners (allOf(), anyOf())

Every task in the tree has another suspended task waiting for its completion

Exceptions propagate up the tree

Cancellation propagates down the tree
Giving it a shot: a simple TCP echo server

Task<void> serve(tcp::socket s) {
    std::array<char> buf(1024);
    try {
        for (;;) {
            size_t len = co_await s.async_read_some(
                asio::buffer(buf), asio::use_awaitable);
            co_await async_write(s, asio::buffer(buf, len),
                asio::use_awaitable);
        }
    } catch (std::exception&) { /*connection closed or I/O error*/ }
}

Task<void> listen(tcp::acceptor& acc) {
    for (;;) {
        tcp::socket s = co_await acc.async_accept(io_context,
            asio::use_awaitable);
        // ???
    }
}
Giving it a shot: a simple echo server

```cpp
Task<void> serve(tcp::socket s) {
    std::array<char> buf(1024);
    try {
        for (;;) {
            size_t len = co_await s.async_read_some(
                asio::buffer(buf), use_awaitable);
            co_await async_write(s, asio::buffer(buf, len),
                                 asio::use_awaitable);
        }
    } catch (std::exception&) { /*connection closed or I/O error*/ }
}

Task<void> listen(tcp::acceptor& acc) {
    tcp::socket s = co_await acc.async_accept(io_context,
                                              asio::use_awaitable);
    co_await anyOf(serve(std::move(s)), accept(acc));
}
```
listen() 🌐

allOf

serve(s1) 🌐 listen()

allOf

serve(s2) 🌐 listen() 📦️ accept()
Dynamic allOf()?
Dynamic allOf()?
Dynamic allOf()?
Dynamic allOf()?

Task<void> serve(tcp::socket);

Task<void> listen(tcp::acceptor& acc) {
    DynamicAllOf tasks;

    for (; ; ) {
        tcp::socket s =
            co_await acc.async_accept(
                io_context, use_awaitable);
        tasks.add(serve(move(s)));
    }
}

This runs unsupervised => ground rule violated
Dynamic allOf()

```cpp
Task<void> serve(tcp::socket);

Task<void> listen(tcp::acceptor& acc) {
    co_await dynamicAllOf(
        [&](DynamicAllOf& tasks) -> Task<void> {
            for (;;) {
                tcp::socket s =
                    co_await acc.async_accept(
                        io_context, use_awaitable);

                tasks.add( serve(move(s)) );
            }
        });
}
```
Task<void> serve(tcp::socket);

Task<void> listen(tcp::acceptor& acc) {
    co_await dynamicAllOf(
        [&](DynamicAllOf& tasks) -> Task<void> {
            for (;;) {
                tcp::socket s =
                    co_await acc.async_accept(
                        io_context, use_awaitable);
                tasks.add(serve(move(s)));
            }
        });
}
Dynamic allOf()

```cpp
co_await dynamicAllOf(
    [&](DynamicAllOf& tasks) -> Task<void> {
        for (;;) {
            tcp::socket s =
                co_await acc.async_accept(
                    io_context, use_awaitable);
            tasks.add(serve(move(s)));
        }
    });
```
Dynamic allOf(

```cpp
co_await dynamicAllOf(
    [&](DynamicAllOf& tasks) -> Task<void> {

    We have a *named node* in our task tree

    Maybe add a bunch of functions there?
    ```add()```
    ```size()```
    ```maybe cancel()```

    No longer a simple allOf()
Dynamic allOf()

We have created a fundamental primitive of structured concurrency

- *nursery* in Python trio
- *task group* in Python asyncio
- *task scope* in Rust
- *coroutine scope* in Kotlin

We also call it a *nursery*
(mnemonic: a place where your children are)
Dynamic allOf()

Task<\texttt{void}> serve(tcp::socket);

Task<\texttt{void}> listen(tcp::acceptor& acc) {

    \texttt{co\_await} openNursery(
        [&](Nursery& nursery) -> Task<\texttt{void}> {

            for (;;) {
                tcp::socket s =
                    \texttt{co\_await} acc.async_accept(
                        io_context, use_awaitable);
                n.start(serve(move(s)));
            }
        })
;

}

}
Nurseries

Task<\texttt{void}> serve(tcp::socket);

Task<\texttt{void}> listen(tcp::acceptor& acc) {
    WITH_NURSERY(nursery) {
        for (; ;) {
            tcp::socket s =
                co\_await acc.async_accept(
                    io\_context, use\_awaitable);
            nursery.start(serve(move(s)));
        }
    }
}
Properties of nurseries and combiners

Act as nodes in the task tree

Wait until all children complete

Propagate any exceptions from any children back to the parent

Cancel any children still running before completing

No task is ever left behind
Bending the rules

What if you need to spawn a coroutine which will outlive the spawner?

What if the spawner is not even a coroutine?

Pass the spawner a reference to a nursery
Bending the rules

```cpp
void beginListen(Nursery& n, io_context& io, uint16_t port) {
    n.start([&](){ -> Task<void> {
        tcp::acceptor acc(io, tcp::endpoint(tcp::v4(), port));
        for (;;) {
            auto s = co_await acc.async_accept(io, use_awaitable);
            n.start(serve(move(s));
        }
    }});
}
```
Bending the rules – to a degree

Every task still has a caller
  • the caller is not what called us – it’s what we’re going to return control to
  • the caller will get any unhandled exceptions

Some room for dangling references
  • passing nurseries around is an advanced technique that attracts extra scrutiny

Task lifetime is still bounded by nursery lifetime

Function behavior is deduced from its signature
Bending the rules – to a degree

Function behavior is deduced from its signature

- `void func()` cannot spawn any child coroutines
- `Task<void> func()` can spawn coroutines, but will join them before returning
- `void func(Nursery&)` can (and likely will) spawn coroutines
- `Task<void> func(Nursery&)` that might outlive it (beware!)
Active objects

class ProcessSupervisor {
public:
    void start(const string& cmdline);

private:
    // suspends until the process completes
    Task<void> runProcess(const string& cmdline);
};
Active objects

class ProcessSupervisor {
    Nursery* nursery = nullptr;

public:
    void start(const string& cmdline) {
        nursery->start(runProcess(cmdline));
    }

private:
    // suspends until the process completes
    Task<void> runProcess(const string& cmdline);
};
Active objects

class ProcessSupervisor {
    Nursery* nursery = nullptr;

How to initialize ProcessSupervisor::nursery?

An obvious approach:
accept it as an argument in class constructor
Active objects

An obvious (and wrong) approach: accept it as an argument in class constructor

```csharp
Task<void> workWithSupervisor() {
    WITH_NURSERY(n) {
        ProcessSupervisor ps(n);
        ps.start("/bin/true");
        // ...stuff...
    };
}
```

supervisor goes out of scope here but tasks spawned in the nursery continue running
Good old two-phase initialization to the rescue

class ProcessSupervisor {
    Nursery* nursery = nullptr;
public:
    Task<void> run() {
        WITH_NURSERY(n) {
            nursery = &n;
            co_await SuspendForever{};
        }
    }
};

Task<void> workWithSupervisor() {
    ProcessSupervisor s;
    co_await anyOf(
        s.run(),
        [&]() -> Task<> {
            s.start("true");
            // ...stuff...
        });
}
Awaitable interface

And implementing your own awaitables
Awaitable interface

```cpp
void await_suspend(std::coroutine_handle<> h)
    Initiate the asynchronous operation, and arrange h.resume() to be called when it completes

auto await_resume()
    Fetch the result of the operation, or (re)throw an exception.

bool await_ready() const noexcept
    Performance optimization
```
Awaitable example

typedef void (*ares_addrinfo_callback)(void* arg, int status,
  int timeouts,
  struct ares_addrinfo* result);

void ares_getaddrinfo(ares_channel_t*,
  const char* name, const char* service,
  const struct ares_addrinfo_hints* hints,
  ares_addrinfo_callback, void* arg);
Awaitable example

class DNSQuery {
    ares_channel_t* channel;
    const char* name;
    struct sockaddr_in result;
    int status;
    std::coroutine_handle<> parent;

public:
    void await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h);
    struct sockaddr_in await_resume();
};
Awaitable example

```cpp
void DNSQuery::await_suspend(std::coroutine_handle<> h) {
    parent = h;
    ares_getaddrinfo(
        channel, name, /*service=*/nullptr, /*hints=*/nullptr,
        [](void* arg, int status, int /*timeouts*/,
           struct ares_addrinfo* ai)
        {
            auto self = static_cast<DNSQuery*>(arg);
            // Copy result back into the awaitable object
            self->status = status;
            if (ai) {
                memcpy(self->result, ai->nodes->ai_addr,
                       sizeof(struct sockaddr_in));
            }
            // Resume the parent
            self->parent.resume();
        }, this);
    }
}
Awaitable example

```cpp
struct sockaddr_in DNSQuery::await_resume() {
    if (status == ARES_SUCCESS) {
        return result;
    }
    else {
        throw std::runtime_error(std::format(
            "cannot resolve {}: {}", name, ares_strerror(status)));
    }
}
```
Awaitable example

Any operation modeled as

```cpp
void beginThing(std::function<void()> doneCB);
```

can be rewritten as an awaitable with

```cpp
void await_suspend(std::coroutine_handle<> h) {
  beginThing([h]{ h.resume(); });
}
```
Task cancellation
Cancellation properties

**Implicit:** no if (cancelled) return scattered throughout code

**Fast:** no C++ exceptions involved

**Asynchronous**
Cancellation properties: *asynchronous*

Some async operations are (or can be made) immediately cancellable

Some lack cancellation support in its API

```c
void begin_thing(void (*callback)(result_t, void* cookie),
                 void* cookie);
```

Some operations inherently cannot be cancelled synchronously
Cancellation properties: *asynchronous*

Some operations inherently cannot be cancelled synchronously

- a running subprocess

  you can SIGKILL it, but you still need to wait() for its completion

```cpp
task< void > execInChroot(string_view chrootPath,
                           string_view cmdline) {
    copyChrootEnv();
    auto procMnt = BindMount("/dev", format("{}/dev", chrootPath));
    co_await runAsync(format("chroot {} sh -c '{}'",
                             chrootPath, cmdline));
    // implicit procMnt.~BindMount()
}
```

This needs to complete before this can unmount
Cancellation properties: *asynchronous*

Some operations inherently cannot be cancelled synchronously

- io_uring or overlapped I/O

```c
BOOL ReadFileEx(HANDLE file, void* buf, DWORD len, OVERLAPPED*, void (*)(DWORD err, DWORD len, OVERLAPPED*));

BOOL CancelIoEx(HANDLE file, OVERLAPPED*);
```

Cancellation happens asynchronously

Callback *will* be called, delivering STATUS_CANCELED or completion result

[https://devblogs.microsoft.com/oldnewthing/?p=11613](https://devblogs.microsoft.com/oldnewthing/?p=11613)
Cancellation properties: *asynchronous*

Some operations inherently cannot be cancelled synchronously

If we only support synchronous cancellation, we are opting out from ever supporting any of these

Rust developers learned that the hard way

https://carllerche.com/2021/06/17/six-ways-to-make-async-rust-easier/
Sketching cancellation procedure

std::coroutine_handle<void>::destroy()

Destroys the suspended coroutine frame and any arguments and local variables of the coroutine still in scope
Cancellation procedure: Phase 2

We need cooperation from the awaitable

```cpp
bool await_ready() const noexcept;
void await_suspend(std::coroutine_handle<> h);
auto await_resume();
```
Cancellation procedure: Phase 2

We need cooperation from the awaitable

```cpp
bool await_ready() const noexcept;
void await_suspend(std::coroutine_handle<> h);
auto await_resume();
void await_cancel(std::coroutine_handle<> h) noexcept;
```

Requests cancellation of the in-progress asynchronous operation.
Upon confirmation, should invoke `h.resume()`.
Handle will match passed to `await_suspend()`, so `void await_cancel(auto) {}` is a legit implementation.
Cancellation procedure: Phase 2

1. Cancellation requested
2. Mark pending cancellation
3. await_cancel()

...time passes...

4. Coroutine resumed
5. Cancellation pending?
6. Resume execution
7. handle.destroy()
We need access to the awaitable

A suspended coroutine knows neither the awaitable nor its type

We need to somehow stash it before suspending

`Promise::await_transform()` to the rescue

Any awaitable is passed through `await_transform()`
We can use it to stash a type-erased pointer to the awaitee
class Promise {
    void* awaitee_ = nullptr;
    void (*cancelFn_)(void*,
        coroutine_handle<>)
        = nullptr;
    bool cancelling_ = false;

template<class T>
    auto await_transform(T&& t) {
        return Interceptor<T>{this, &t};
    }

template<class T> struct Interceptor {
    Promise* p;
    T* t;

    auto await_suspend(coroutine_handle<> h) {
        // Stash a type-erased reference
        // to the awaitee
        p->awaitee_ = t;
        p->cancelFn_ = +[](void* ptr,
            coroutine_handle<> h){
            reinterpret_cast<T*>(ptr)
                ->await_cancel(h);
        };
        return t->await_suspend(h);
    }

decltype(auto) await_resume() {
    p->awaitee_ = nullptr;
    p->cancelFn_ = nullptr;
    return forward<T>(*t).await_resume();
}

bool await_ready() const noexcept {
    return t->await_ready();
}

    void cancel() {
        cancelling_ = true;
        if (awaitee_) {
            cancelFn_(awaitee_,
                coroutine_handle<Promise>::from_promise(*this));
        }
    }
};
Destroying the coroutine

coroutine_handle::destroy() can only be called on a suspended coroutine

Once the awaitee resume()d the coroutine handle, the coroutine is no longer suspended

Awaitables cannot do anything else to the coroutine handle
A coroutine frame

```c
struct coroutine_frame {
    // resume index
    // promise object
    // ...parameters...
    // ...local variables...
    // ...temporary objects...
    // other bookkeeping
};
```
A coroutine frame

```c
struct coroutine_frame {
    void (*resume_fn)(coroutine_frame*);
    void (*destroy_fn)(coroutine_frame*);

    // resume index
    // promise object
    // ...parameters...
    // ...local variables...
    // ...temporary objects...
    // other bookkeeping
};
```
A coroutine frame

```c
struct coroutine_frame {
    void (*resume_fn)(coroutine_frame*);
    void (*destroy_fn)(coroutine_frame*);
    // ...
};
```
A coroutine frame

```cpp
struct coroutine_frame {
    void (*resume_fn)(coroutine_frame*);
    void (*destroy_fn)(coroutine_frame*);
    // ...
};

void coroutine_handle<void>::resume() {
    auto f =
        reinterpret_cast<coroutine_frame*>(address());
    f->resume_fn(f);
}
```
A phony coroutine frame

```c++
struct CoroutineHeader {
    void (*resume_fn)(CoroutineHeader*);
    static coroutine_handle<> toHandle() {
        return coroutine_handle<>::from_address(this);
    }
};

class Promise: private CoroutineHeader {
    auto realHandle() {
        return coroutine_handle<Promise>::from_promise(*this);
    }
    auto proxyHandle() { return CoroutineHeader::toHandle(); }
    void onResum();

public:
    Promise() {
        this->resume_fn = [] (void* self) {
            static_cast<Promise*>(self)->onResum();
        };
    }
};
```
A phony coroutine frame

class Promise: private CoroutineHeader {
    auto realHandle() {
        return coroutine_handle<Promise>::from_promise(*this);
    }
    auto proxyHandle() { return CoroutineHeader::toHandle(); }  
    void onResume();
public:
    Promise() {
        this->resume_fn = [](void* self) {
            static_cast<Promise*>(self)->onResume();
        };
    }
};
A phony coroutine frame

class Promise: private CoroutineHeader {
    auto realHandle();
    auto proxyHandle();

template<class T> struct Interceptor {
    Promise* p;
    T* t;

    auto await_suspend(coroutine_handle<>) {
        // populate awaitee_ and cancelFn_
        return t->await_suspend(p->proxyHandle());
    }
};

void onResume() {
    if (cancelling_) {
        realHandle().destroy();
    } else {
        realHandle().resume();
    }
}
Rejected cancellation

A cancelled awiatable may complete successfully and deliver a result

That result may or may not be safely destroyed

```c
int fd = co_await asyncOpen("/mnt/nfs/remote/file");
```

An awiatable may need an option to request resumption of the parent
Rejected cancellation

Expanding awaitable interface again

```cpp
bool await_ready() const noexcept;
void await_suspend(std::coroutine_handle<> h);
auto await_resume();
void await_cancel(std::coroutine_handle<> h);

bool await_must_resume() const noexcept;
```

Called after resuming an awaiter pending cancellation. If returns `false`, parent will be destroyed (`await_resume()` will not be called); otherwise resumed normally.
Notes on implementing awaitables

Paired functions: constructor/destructor and `await_suspend()/await_cancel()`

`await_cancel()` reverses `await_suspend()`, destructor reverses constructor

Ideally an awaitable object should be dormant until `await_suspend()`
   Otherwise, need to account for completion before `await_suspend()`
   Also need to account for early cancellation
Resource management
RAII even more important than before

```cpp
// don't do this
Task< void > bad() {
    int fd = ::open("/etc/passwd", O_RDONLY);
    std::array<char, 65536> buf;
    ssize_t len = ::read(fd, buf.data(), buf.size());
    co_await publishOnFacebook(buf.data(), len);
    ::close(fd);
}
```
RAII even more important than before

// don't do this either
Task<void> also_bad() {
    int fd = ::open("/etc/passwd", O_RDONLY);
    try {
        std::array<char, 65536> buf;
        ssize_t len = ::read(fd, buf.data(), buf.size());
        co_await publishOnFacebook(buf.data(), len);
        ::close(f);
    }
    catch (...) {
        ::close(f);
        throw;
    }
}
RAII even more important than before

// maybe do this
Task<void> better() {
    int fd = ::open("/etc/passwd", O_RDONLY);
    auto _ = gsl::finally([fd]{ ::close(fd); });

    std::array<char, 65536> buf;
    ssize_t len = ::read(fd, buf.data(), buf.size());
    co_await publishOnFacebook(buf.data(), len);
}

// better yet, use std::ifstream
Asynchronous resource cleanup

```c++
struct AsyncFile {
    static Task<AsyncFile> open(fs::path);
    Task<void> close();
    Task<ssize_t> readInto(span<byte> buffer);
};

Task<void> func() {
    auto f = co_await AsyncFile::open("/etc/passwd");
    // ...work with file...
    std::array<char, 1024> buf;
    ssize_t len = co_await f.readInto(buf);

    // bad idea
    co_await f.close();
}
```
Asynchronous resource cleanup

```c++
struct AsyncFile {
    static Task<AsyncFile> open(fs::path);
    Task<void> close();
    Task<ssize_t> readInto(span<byte> buffer);
};

Task<void> func() {
    auto f = co_await AsyncFile::open("/etc/passwd");
    // a better idea, but does not work
    auto _ = gsl::finally([&f]{ co_await f.close(); });

    // ...work with file...
    std::array<char, 1024> buf;
    ssize_t len = co_await f.readInto(buf);
}
```
Asynchronous resource cleanup

struct AsyncFile {
    static Task<AsyncFile> open(fs::path);
    Task<void> close();
    Task<ssize_t> readInto(span<byte> buffer);
};

Task<void> func() {
    auto f = co_await AsyncFile::open("/etc/passwd");
    co_await anyOf([&]() -> Task<> {
        // ...work with file...
        std::array<char, 1024> buf;
        ssize_t len = co_await f.readInto(buf);
    }, untilCancelledAnd(f.close()));
}
Executors and interruption points
Let's write an Event

A very basic synchronization primitive

Allows one-time, one way transition from not-triggered to triggered state
Let's write an Event

class Event {
    class Awaitable;
    std::set<Awaitable*> awaitables;
    bool triggered;

public:
    void trigger();
    auto operator co_await();
};

class Event::Awaitable {
    Event* evt;
    coroutine_handle<>awaiter;

public:
    explicit Awaitable(Event* e):
        evt(e) {}

    bool await_ready() const noexcept {
        return evt->triggered;
    }
}

void await_suspend(coroutine_handle<> h) {
    awaiter = h;
    evt->awaitables.insert(this);
}

void await_cancel(coroutine_handle<> h) {
    evt->awaitables.erase(this);
    h.resume();
}

void await_resume() {}

bool await_must_resume() const noexcept {
    return false;
}

auto Event::operator co_await() {
    return Awaitable{this};
}

void Event::trigger() {
    triggered = true;
    auto set = std::move(awaitables);
    for (Awaitable* a: set) {
        set->awaiter.resume();
    }
}
Giving it a shot

Event evt;

co_await allOf([&]() -> Task<> { 
    cout << "a: waiting for event\n";
    co_await evt;
    cout << "a: done waiting\n";
}, [&]() -> Task<> {
    co_await sleepFor(100ms);
    cout << "b: setting event\n";
    evt.set();
    cout << "b: done setting\n";
});
Giving it a shot

Event evt;

co_await allOf([&]() -> Task<> {  
cout << "a: waiting for event\n";
co_await evt;
cout << "a: done waiting\n";
}, [&]() -> Task<> {  
co_await sleepFor(100ms);
cout << "b: setting event\n";
evt.set();
cout << "b: done setting\n";
});
Local reasoning at risk

Event evt;
auto th = make_unique<Thing>;
co_await allOf([&]() -> Task<> {
    co_await evt;
    th = nullptr; // evil laugh
}, [&]() -> Task<> {
    co_await sleepFor(100ms);
    poorVictim(th);
});

void somethingTotallyUnrelated() {
    evt.trigger();
}

void poorVictim(unique_ptr<Thing>& th) {
    assert(th);
    somethingTotallyUnrelated();
    cout << th->name();
}
We need another ground rule

Each coroutine runs uninterrupted until its next `co_await`
class Promise: private CoroutineHeader {
    auto proxyHandle();
    auto realHandle();

    void onResume() {
        if (cancelling_) {
            realHandle().destroy();
        } else {
            realHandle().resume();
        }
    }
};
We need an executor

class Promise: private CoroutineHeader {
  auto proxyHandle();
  auto realHandle();

  void onResume() {
    if (cancelling_) {
      realHandle().destroy();
    } else {
      executor_.defer([h = realHandle()] { h.resume(); });
    }
  }
};
We need an executor, let's write one

class Executor {
    queue<coroutine_handle<> > queue_;  
    bool running_ = false;

    public:
    void runSoon(coroutine_handle<> h) {
        queue_.push(h);
        if (!running) {
            running = true;
            while (!queue_.empty()) {
                auto h = queue_.top();
                queue_.pop();
                h.resume();
            }
            running_ = false;
        }
    }

    thread_local Executor* g_executor;

class Promise {
    void onResume() {
        if (cancelling_) {
            realHandle().destroy();
        } else {
            g_executor->runSoon(
                realHandle());
        }
    }
};
Top-level coroutine

template<class EventLoop>
struct EventLoopTraits {
    void run(EventLoop&);
    void stop(EventLoop&);
};

auto run(auto& eventLoop, auto&& awaitable);
    Starts *awaitable* and runs *eventLoop* until it completes.
Bridging to legacy code
Bridging to legacy code

Most code is likely old-style

Coroutine part needs to cooperate
Coroutines to callbacks

```cpp
void beginThing(std::function<void(int /*result*/)> cb);
Task<void> workWithThing() {
    int x = co_await beginThing(/*???*/);
}
```
Coroutines to callbacks: wrap to the awaitable

```cpp
void beginThing(std::function<void(int /*result*/)> cb);

class Thing {
    std::coroutine_handle<> parent;
    int result;

public:
    bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h) {
        parent = h;
        beginThing([this](int res) {
            result = res;
            parent.resume();
        });
    }
    int await_resume() { return result; }
};

Task<void> workWithThing() {
    int x = co_await Thing{};
}
```
Callbacks to coroutines

```cpp
struct ILegacyReader {
    virtual void read(span<byte> dst, function<void(ssize_t)> cb) = 0;
};

class OurReader: public ILegacyReader {
    public:
        void read(span<byte> dst, function<void(ssize_t)> cb) override;
    private:
        Task<size_t> doRead(span<byte> dst);
};
```
Callbacks to coroutines

class OurReader: public ILegacyReader {
public:
    void read(span<byte> dst, function<void(ssize_t)> cb) override;
private:
    Task<size_t> doRead(span<byte> dst);
};
Callbacks to coroutines

class OurReader: public ILegacyReader {
    Nursery* nursery = /*...*/;

public:
    void read(span<byte> dst, function<void(ssize_t)> cb) override {
        nursery->start([=]() -> Task {
            size_t ret = co_await doRead(dst);
            cb(ret);
        });
    }

private:
    Task<size_t> doRead(span<byte> dst);
};
Callbacks to coroutines

class OurReader: public ILegacyReader {
    Nursery* nursery = /*...*/;

public:
    void read(span<byte> dst, function<void(ssize_t)> cb) override {
        nursery->start([=]() -> Task {
            try {
                size_t ret = co_await doRead(dst);
                cb(ret);
            } catch (std::exception&) {
                cb(-1);
            }
        });
    }

private:
    Task<size_t> doRead(span<byte> dst);
};
Callbacks to coroutines

```cpp
class OurReader: public ILegacyReader {
    Nursery* nursery = nullptr;

public:
    void read(span<byte> dst, function<void(ssize_t)> cb) override;
    Task<void> run() {
        WITH_NURSERY(n) {
            nursery = &n;
            co_await SuspendForever{};
        }
    }

private:
    Task<size_t> doRead(span<byte> dst);
};
```
Unsafe nurseries

class OurReader: public ILegacyReader {
    UnsafeNursery nursery;
public:
    void read(span<byte> dst, function<void(ssize_t)> cb) override {
        nursery.start([=]() -> Task {
            try {
                size_t ret = co_await doRead(dst);
                cb(ret);
            } catch (std::exception&) {
                cb(-1);
            }
        });
    }
private:
    Task<size_t> doRead(span<byte> dst);
};
Unsafe nurseries: back to slide one

Unhandled exceptions have nowhere to go

`terminate()` if we got any

Destructor must not leave running coroutines behind

make one attempt to cancel any tasks still alive

if that did not help, `terminate()`

Bridging different paradigms is toilsome anyway
Our own experience

First version merged in Dec 2021

Used in several I/O-bound services
Our own experience

About 40-50% code size reduction

Easier control flow

Uniform handling of cancellation and timeouts

There is a learning curve

There are performance implications
Our own experience

About 40-50% code size reduction
Easier control flow
Uniform handling of cancellation and timeouts
There is a learning curve
There are performance implications

github.com/hudson-trading/corral
Thank you!

Questions?

github.com/hudson-trading/corral