Automatically Process Your Operations in Bulk With Coroutines

FRANCESCO ZOFFOLI

Cppcon
The C++ Conference
2021
October 24-29

About me

- Professional software engineer
- Passionate about C++, used throughout career
- Currently building monitoring systems at Facebook
- Author of the book “C++ Fundamentals” - Packt

makers.f.dev@gmail.com

github.com/MakersF

Presentation

- The Problem
- The Idea
- Coroutines in C++20
- The Implementation
- Benchmark

Presentation

Code on github: github.com/MakersF/cppcon-2021-corobatch

The Problem

The Problem

Bulk operations tend to be better than single operations

Bulk operation: processing multiple actions or data (a batch) at once

The Problem

- networking
- file I/O
- memory allocation
- GPU draw calls
- ...

Common property:

cost = FixedCost + O(#operation/data)

The Problem

Networking latency

round trip ≫ data transfer

Write disk

disk seek ≫ data write

DB table scan

loading data in memory ≫ conditions to check per row

What’s better?

Multiple aspect

- throughput
- latency
- others: rate limit, cost, …

Different properties are important in different situations

The Problem

Use case:

“Send a notification to a group of users, respecting their preferences”

The Problem

Bulk operations

- fetching preferences
- sending notifications

Signature

```
vector<UserPrefs> getUserPrefs(vector<UserId> userIds);
vector<bool> sendNotifs(vector<EmailAddress> addresses);
```

Example - No Batching

```
for(const User& user : users) {
  const UserPrefs prefs = getUserPrefs({user.id}).at(0);
  if (prefs.wantsEmailNotification) {
    sendNotifs({prefs.notificationEmail});
  }
}
```

The Problem

But batching is important!

Example - Batching

```
vector<UserId> userIds;
transform(users.begin(), users.end(),
          back_inserter(userIds),
          [](auto& user) { return user.id; });
vector<UserPrefs> preferences = getUserPrefs(userIds);
```

Example - Batching

```
remove_if(preferences.begin(), preference.end(),
          [](auto& pref) { return !pref.wantsEmailNotification;});
```

Example - Batching

```
vector<EmailAddress> emails;
transform(preferences.begin(), preference.end(),
          back_inserter(emails),
          [](auto& pref) { return pref.notificationEmail; });
sendNotification(emails);
```

Example - Batching

```
vector<UserId> userIds;
transform(users.begin(), users.end(),
          back_inserter(userIds),
          [](auto& user) { return user.id; });
vector<UserPrefs> preferences = getUserPrefs(userIds);

remove_if(preferences.begin(), preference.end(),
          [](auto& pref) { return !pref.wantsEmailNotification;});

vector<EmailAddress> emails;
transform(preferences.begin(), preference.end(),
          back_inserter(emails),
          [](auto& pref) { return pref.notificationEmail; });
sendNotification(emails);
```

Readability

Code is different

- requires effort
- extra memory
- cannot use const

Benchmark

Simulation

- 10ms delay per call
- 50us delay per item
- 100 users
- 50 emails to send

Benchmark

Test	Time (us)	Iterations	Baseline
No Batching	1’552’542	13	49.99
Manual Batching	31’055	492	-

A 50x difference

Questions?

The Idea

The Idea

Can we keep

- the readability of the first version, and
- the batching behaviour of the second?

The Idea

What if we could gather all the parameters passed to the function?

```
for(const User& user : users) {
  const UserPrefs prefs = getUserPrefs({user.id}).at(0);
  if (prefs.wantsEmailNotification) {
    sendNotifs({prefs.notificationEmail});
  }
}
```

The Idea

Imagine we have 4 users

```
for(const User& user : users) {
  const UserPrefs prefs = getUserPrefs({user.id}).at(0);
  if (prefs.wantsEmailNotification) {
    sendNotifs({prefs.notificationEmail});
  }
}
```

The Idea

```
for(user : users) {
  prefs = getUserPrefs(user.id);
  if (prefs.notify) {
    sendNotification(prefs.email);
  }
}
```

User 1

```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```

User 2

```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```

User 3

```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```

•••

User n

```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```

The Idea

User 1

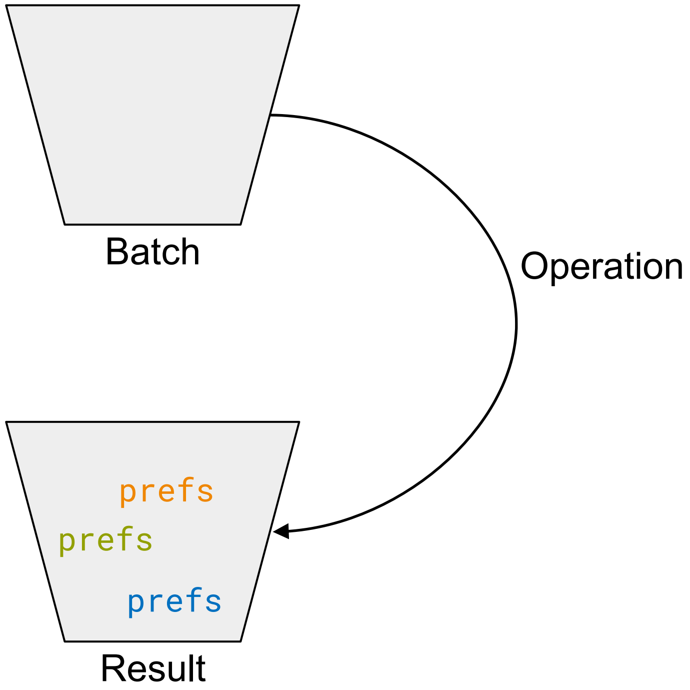
```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```

User 2

```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```

User 3

```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```



The Idea

User 1

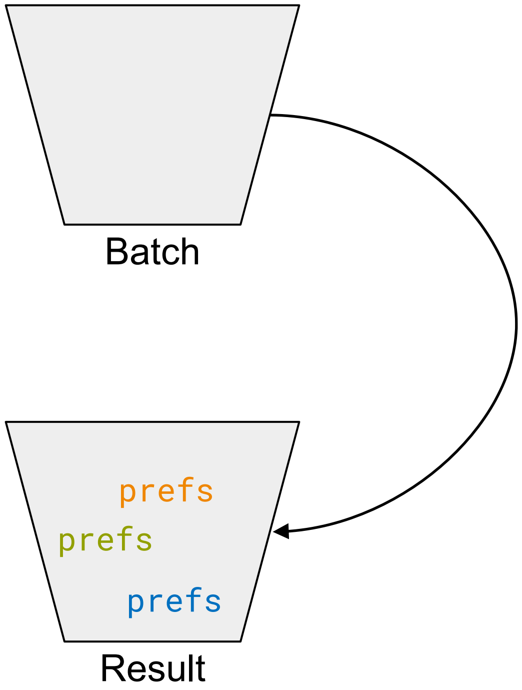
```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```

User 2

```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```

User 3

```
prefs = getUserPrefs(user.id);
if (prefs.notify) {
  sendNotification(prefs.email);
}
```



The Idea

1. Execute until a batch operation
2. Record the argument, and stop
3. If not enough arguments, go to point 1
4. Execute the batch operation
5. Resume the stopped executions with the results

The Idea

What do we need?

- Suspend execution
- Resume execution

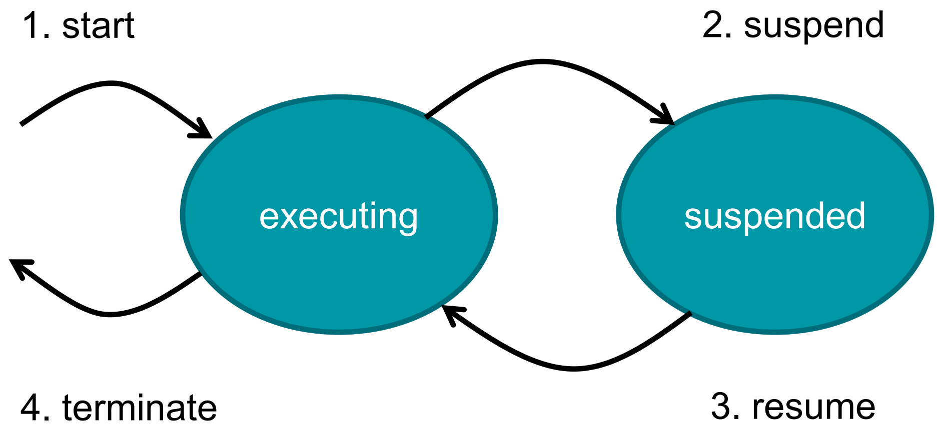
Coroutines allow us to do exactly that

Questions?

Coroutines

Coroutines

Resumable functions



Coroutines in C++

How does a coroutine look like?

```
task my_coro() {
   cout << "foo begin\n";
   co_await async_operation();
   co_return;
}
```

Coroutines in C++

To be a coroutine a function must

- use one of: `co_await` , `co_yield` and `co_return`
- return a type which satifies the requirements for coroutines

Coroutines in C++ - promise

Requirements for the return type

- Must contain a nested type called `promise_type`
- `return_type::promise_type` must implement several functions

Coroutines in C++ - promise

The body is rewritten

```
{
  promise_type promise;
  co_await promise.initial_suspend();
  try {
    [function-body]
  } catch (...) {
    promise.unhandled_exception();
  }
final_suspend:
  co_await promise.final_suspend();
}
```

Coroutines in C++ - promise

Let's build a valid coroutine return type

```
struct task {
  struct promise_type {





  };
};
```

Coroutines in C++ - promise

When `my_coro` is invoked, the compiler

1. creates the coroutine state
2. instantiates `task::promise_type`
3. calls `get_return_object`

```
struct task {
  struct promise_type {
    task get_return_object();




  };
};
```

Coroutines in C++ - promise

It calls `initial_suspend()` and `co_await`s the result

```
struct task {
  struct promise_type {
    task get_return_object();
    Awaitable initial_suspend();



  };
};
```

Coroutines in C++ - co_await

What happens in `co_await awaitable`?

The compiler inserts calls to `awaitable` methods

```
struct Awaitable {



};
```

Coroutines in C++ - co_await

First, it calls `await_ready()`

`true` → immediately resumes the coroutine
`false` → suspends the coroutine

```
struct Awaitable {
  bool await_ready();



};
```

Coroutines in C++ - co_await

After suspending the coroutine, it calls `await_suspend()`

Should schedule the coroutine to be resumed

```
struct Awaitable {
  bool await_ready();
  auto await_suspend(std::coroutine_handle<>);

};
```

Coroutines in C++ - co_await

Once resumed, it calls `await_resume()`

The `co_await` expression evaluates to the returned object

```
struct Awaitable {
  bool await_ready();
  auto await_suspend(std::coroutine_handle<>);
  T await_resume();
};
```

Coroutines in C++ - handle

What is the `std::coroutine_handle` we saw a few times?

Coroutines in C++ - handle

A handle

- a not owning pointer
- controls the coroutine

Coroutines in C++ - handle

`std::coroutine_handle<>` methods

- `resume()`: resume the associated coroutine
- `destroy()`: destroy the associated coroutine

```
struct coroutine_handle<> {
  void resume();
  void destroy();




};
```

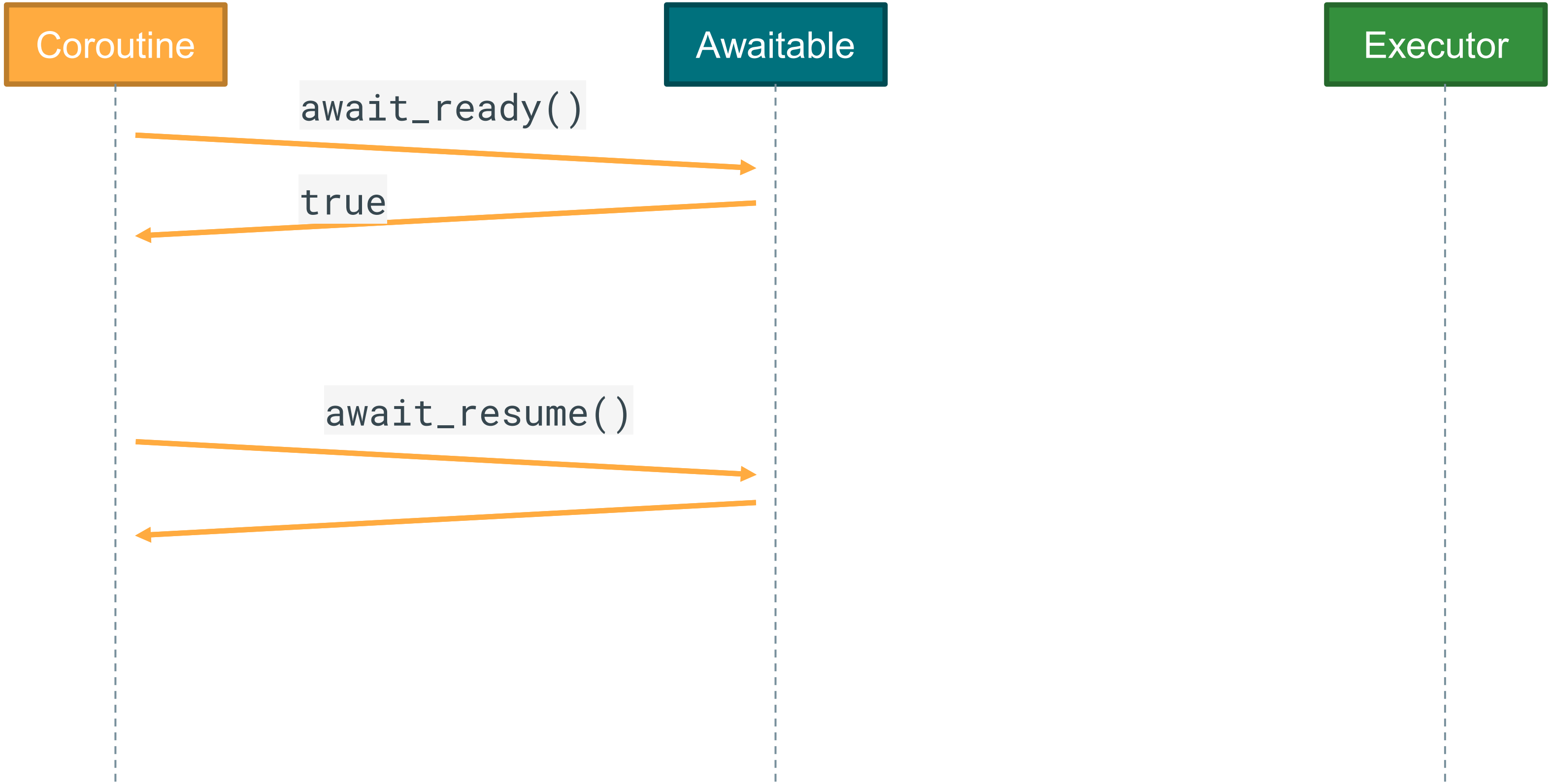
Coroutines in C++ - handle

`std::coroutine_handle<promise_type>` additional methods

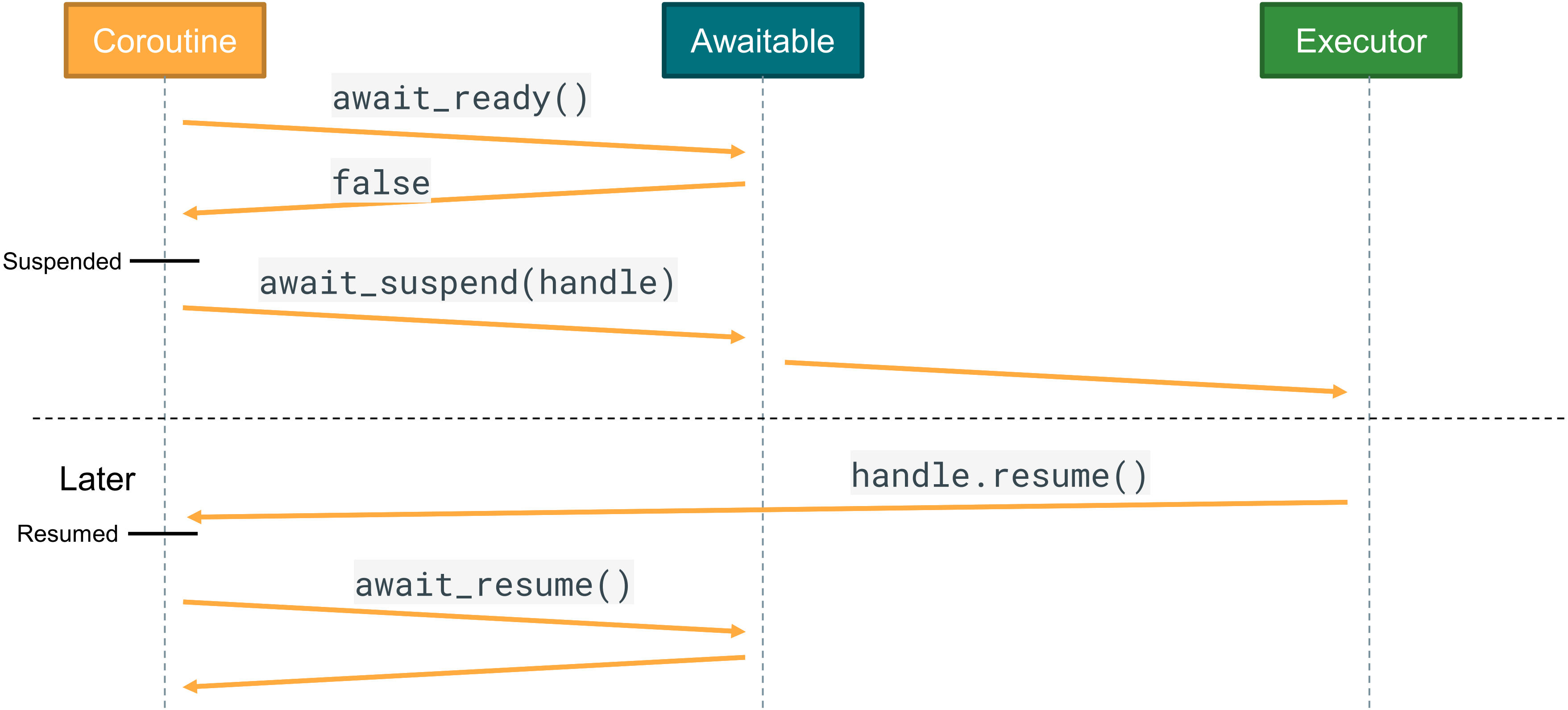
- `promise()`: returns the associated promise
- `from_promise()`: creates a handle from the promise

```
struct coroutine_handle<promise_type> {
  void resume();
  void destroy();
  promise_type& promise();
  static coroutine_handle<promise_type> from_promise(promise_type&);
};
```

Coroutines in C++ - co_await



Coroutines in C++ - co_await



Coroutines in C++ - promise

Back to the promise type!

```
struct task {
  struct promise_type {
    task get_return_object();
    Awaitable initial_suspend();



  };
};
```

Coroutines in C++ - promise

How to return values

Coroutines in C++ - promise

`co_return expr;` → `return_value(expr)`

```
struct task {
  struct promise_type {
    task get_return_object();
    Awaitable initial_suspend();
    void return_value(T);


  };
};
```

Coroutines in C++ - promise

`co_return;` → `return_void()`

```
struct task {
  struct promise_type {
    task get_return_object();
    Awaitable initial_suspend();
    void return_void();


  };
};
```

Coroutines in C++ - promise

Uncaught exceptions → `uncaught_exception()`

```
struct task {
  struct promise_type {
    task get_return_object();
    Awaitable initial_suspend();
    void return_void();
    void uncaught_exception();

  };
};
```

Coroutines in C++ - promise

Finally, `final_suspend()` is called and `co_await`ed

```
struct task {
  struct promise_type {
    task get_return_object();
    Awaitable initial_suspend();
    void return_void();
    void uncaught_exception();
    Awaitable final_suspend() noexcept;
  };
};
```

Coroutines in C++

We know everything to implement the idea!

Questions?

Implementing the Idea

Implementing the Idea

We learned how to

- suspend execution
- resume execution

Implementing the Idea

1. Transform the loop body in a coroutine
2. Use await to interrupt execution, store the argument
3. Execute the batch when ready
4. Resume execution with the result

Implementing the Idea

Before

```
for(const User& user : users) {
  const UserPrefs prefs = getUserPrefs({user.id}).at(0);
  if (prefs.wantsEmailNotification) {
    sendNotifs({prefs.notificationEmail});
  }
}
```

After

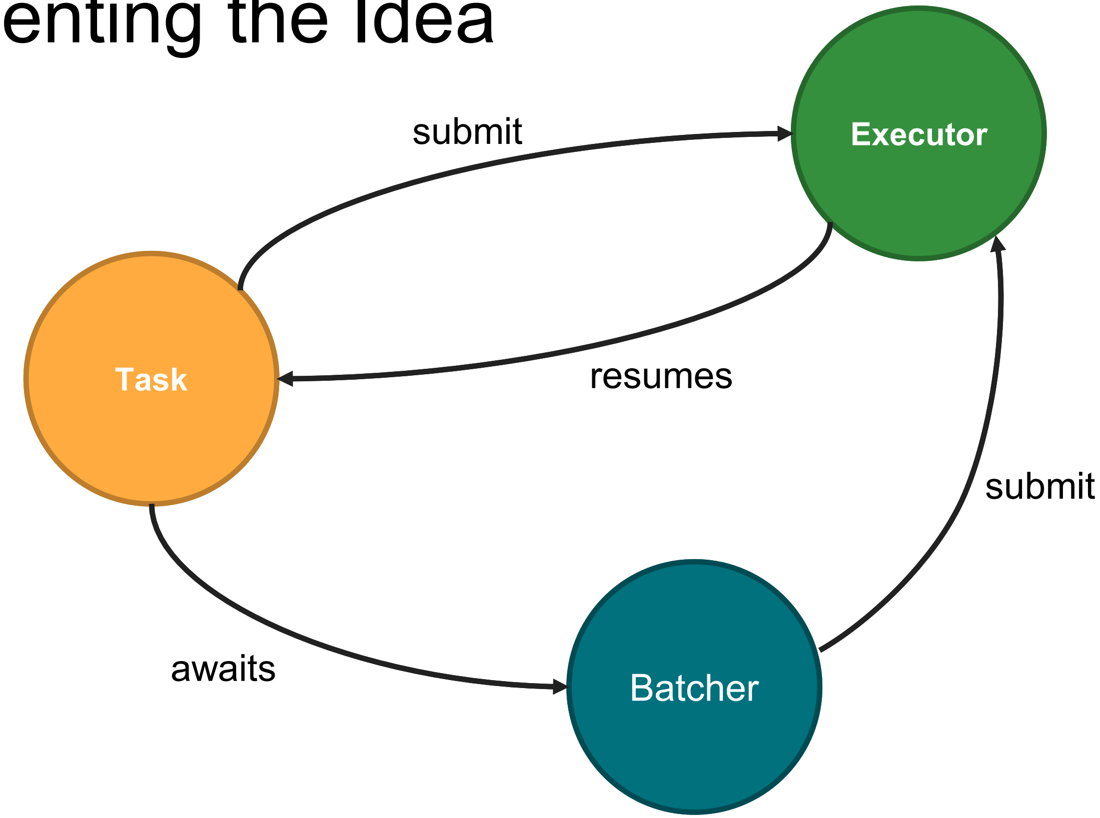
```
task sendEmail(const User& user) {
 const UserPrefs prefs = co_await getUserPrefs(user.id);
  if (prefs.wantsEmailNotification) {
    co_await sendNotifs(prefs.notificationEmail);
  }
}
```

Implementing the Idea

We'll need

1. a task
2. an executor
3. a batcher

Implementing the Idea



Implementing the Idea

```
Executor e;
Batcher<…> getUserPrefs{…};
Batcher<…> sendNotifs{…};

auto sendEmail = [&](const User& user) -> task {
  …
};

for(auto& user : users) {
  e.submit(sendEmail(user));
}

e.run_available();
```

Implementing the Idea - Task

Let's look at the task

```
struct task {






};
```

Implementing the Idea - Task

It must have a `promise_type`

```
struct task {
  struct promise_type { … };
  using Handle = coroutine_handle<promise_type>




};
```

Implementing the Idea - Task

Similar to a `unique_ptr`

Owns the `coroutine_handle`

```
struct task {
  struct promise_type { … };
  using Handle = coroutine_handle<promise_type>
  explicit task(Handle ptr) : ptr_(ptr) {}
  …

private:
  Handle ptr_;
};
```

Implementing the Idea - Task

It’s only moveable

```
struct task {
  …

  task(const task&) = delete;
  task(task&& t) noexcept
    : ptr_(exchange(t.ptr_, nullptr)) {}

  …
};
```

Implementing the Idea - Task

Has RAII semantic

```
struct task {
  …

  ~task() {
    if (ptr_) ptr_.destroy();
  }

  …
};
```

Implementing the Idea - Task

Exposes the `coroutine_handle`

```
struct task {
  …

  Handle release() && {
    return exchange(ptr_, nullptr);
  }


};
```

Implementing the Idea - Task

Let’s look at the `promise_type`

```
struct task::promise_type {




};
```

Implementing the Idea - Task

We immediately return when the coroutine is called

```
struct task::promise_type {
  suspend_always initial_suspend() { return {}; }




};
```

Implementing the Idea - Task

We don’t suspend when the coroutine terminates

```
struct task::promise_type {
  suspend_always initial_suspend() { return {}; }
  suspend_never final_suspend() noexcept { return {}; }



};
```

Implementing the Idea - Task

Ignore exceptions

```
struct task::promise_type {
  suspend_always initial_suspend() { return {}; }
  suspend_never final_suspend() noexcept { return {}; }
  void unhandled_exception() { terminate(); }


};
```

Implementing the Idea - Task

We do nothing when we return

```
struct task::promise_type {
  suspend_always initial_suspend() { return {}; }
  suspend_never final_suspend() noexcept { return {}; }
  void unhandled_exception() { terminate(); }
  void return_void() {}

};
```

Implementing the Idea - Task

Let’s look at how to implement `get_return_object()`

```
struct task::promise_type {
  suspend_always initial_suspend() { return {}; }
  suspend_never final_suspend() noexcept { return {}; }
  void unhandled_exception() { terminate(); }
  void return_void() {}
  task get_return_object();
};
```

Implementing the Idea - Task

A reminder

```
struct std::coroutine_handle<promise_type> {
  void resume();
  void destroy();
  promise_type & promise();
  static coroutine_handle<promise_type> from_promise(promise_type&);
};
```

Implementing the Idea - Task

```
task get_return_object() {
  Handle handle = Handle::from_promise(*this);
  return task(handle);
}
```

Implementing the Idea - Task

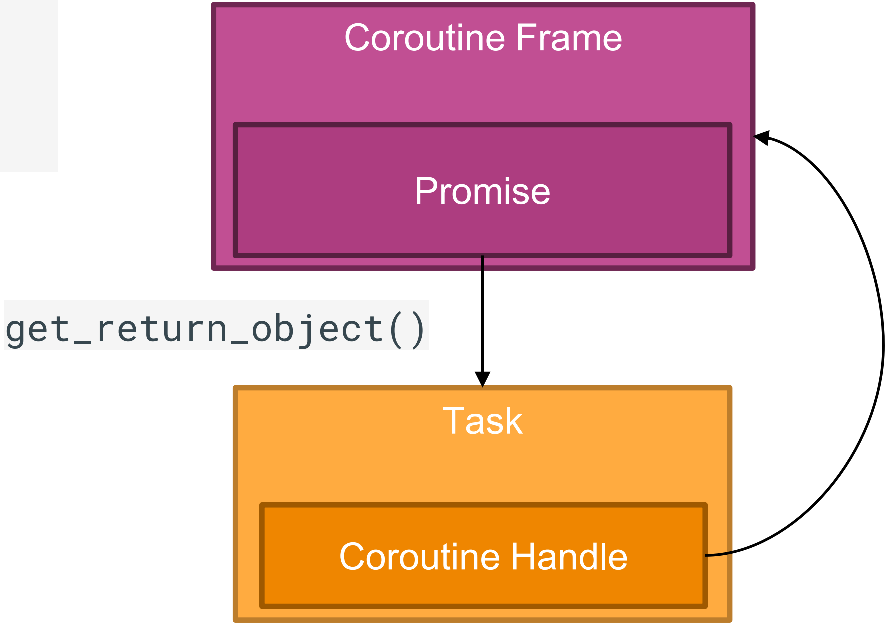
Our `task` is complete!

We can now write

```
task sendEmail(const User& user) {



  co_return;
}
```

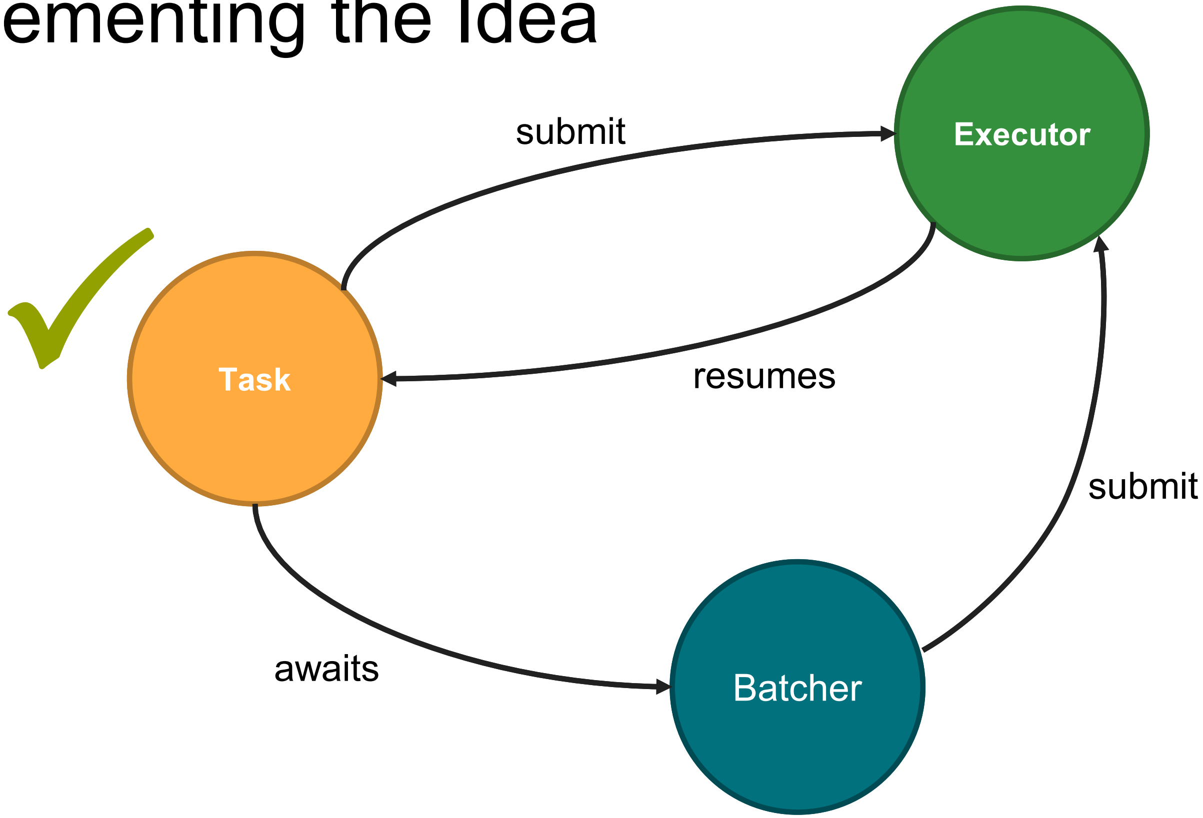
Implementing the Idea - Task

```
task coro = sendEmail(user);
```



Questions?

Implementing the Idea



Implementing the Idea - Executor

We need something to execute the `task`s

Implementing the Idea - Executor

We need something to execute the `tasks`

Let’s create an `Executor`

```
struct Executor {




};
```

Implementing the Idea - Executor

Supports

- submit coroutines
- execute pending coroutines

Implementing the Idea - Executor

Stores pending `coroutine_handle`s

```
struct Executor {





  deque<coroutine_handle<>> pending_;
};
```

Implementing the Idea - Executor

Submitting coroutines

```
struct Executor {







  deque<coroutine_handle<>> pending_;
};
```

Implementing the Idea - Executor

Can submit a `task`

```
struct Executor {
  void submit(task t) {
    auto handle = move(t).release();
    pending_.push_back(handle);
  }




  deque<coroutine_handle<>> pending_;
};
```

Implementing the Idea - Executor

And arbitrary `coroutines`

```
struct Executor {
  void submit(task t) {
    auto handle = move(t).release();
    pending_.push_back(handle);
  }
  void submit(vector<coroutine_handle<>> coros) {
    // insert at the end
  }

  deque<coroutine_handle<>> pending_;
};
```

Implementing the Idea - Executor

Executing coroutines

```
struct Executor {








  deque<coroutine_handle<>> pending_;
};
```

Implementing the Idea - Executor

Pull the first coroutine

```
struct Executor {
  optional<coroutine_handle<>> pop_next_coro() {
    if( pending_.empty() ) return nullopt;
    auto first = pending_.front();
    pending_.pop_front();
    return first;
  }


  deque<coroutine_handle<>> pending_;
};
```

Implementing the Idea - Executor

Pull the first coroutine

```
struct Executor {
  optional<coroutine_handle<>> pop_next_coro();

  void run_available() {
    for(auto next = pop_next_coro(); next; next = pop_next_coro()) {
      next->resume();
    }
  }

  deque<coroutine_handle<>> pending_;
};
```

Implementing the Idea - Executor

Summary

```
struct Executor {
  void submit(task t);
  void submit(vector<coroutine_handle<>> coros);

  optional<coroutine_handle<>> pop_next_coro();
  void run_available();
};
```

Implementing the Idea - Executor

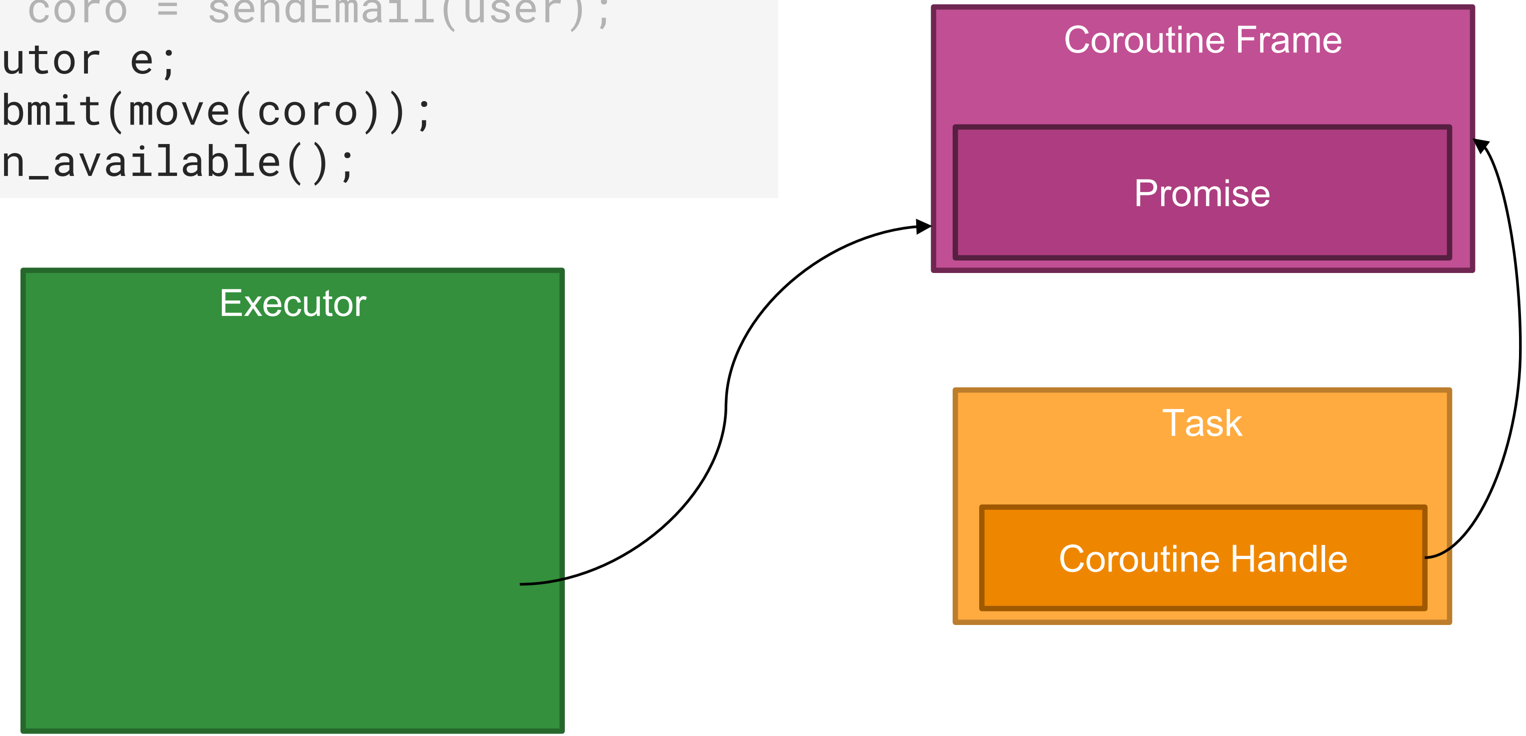
Our `Executor` is complete!

We can now write

```
Executor e;
e.submit(sendEmail(user));
e.run_available();
```

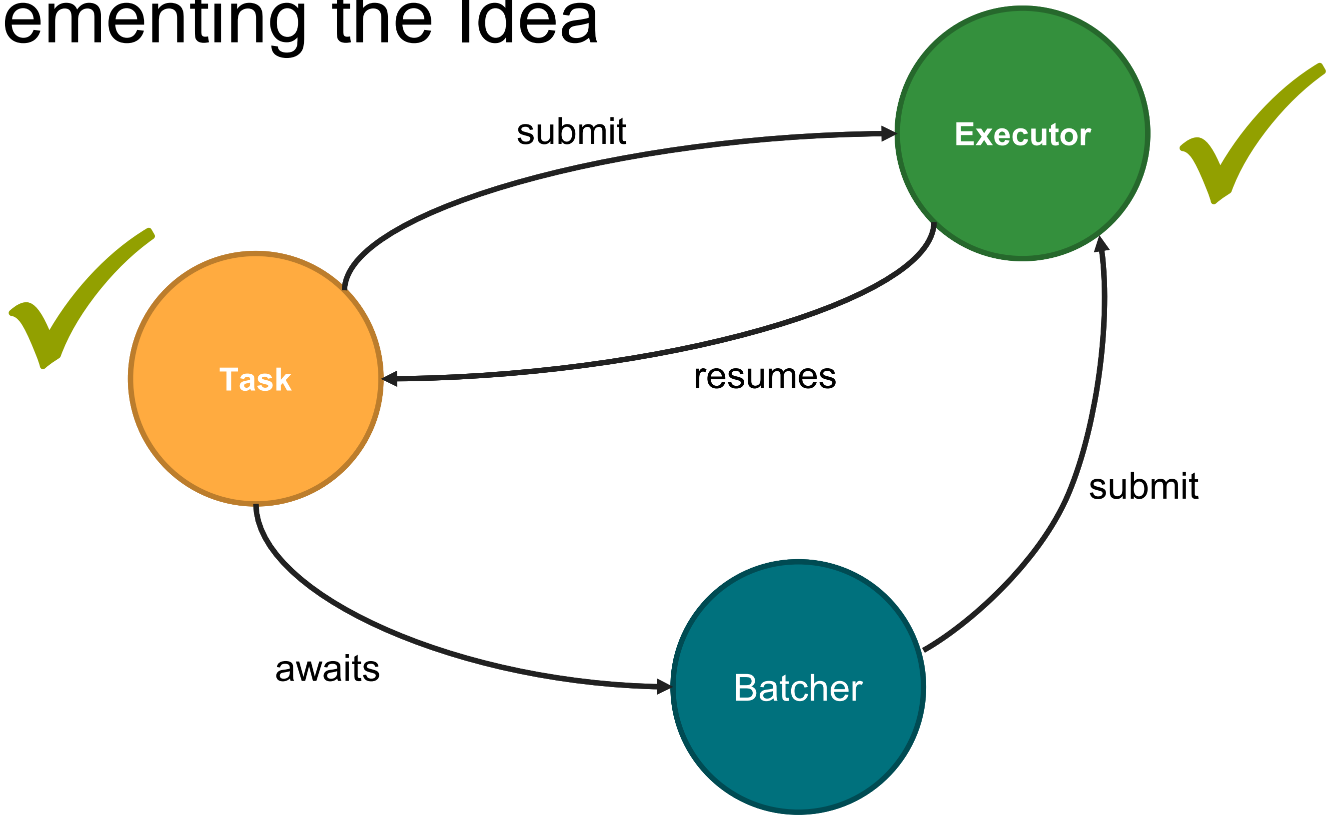
Implementing the Idea - Executor

```
task coro = sendEmail(user);
Executor e;
e.submit(move(coro));
e.run_available();
```



Questions?

Implementing the Idea



Implementing the Idea - Batcher

Let's support `co_await` now!

```
task sendEmail(const User& user) {
 const UserPrefs prefs = co_await getUserPrefs(user.id);



}
```

Implementing the Idea - Batcher

`co_await` must be called on an `awaitable`

`getUserPrefs(user.id)` must return an `awaitable`

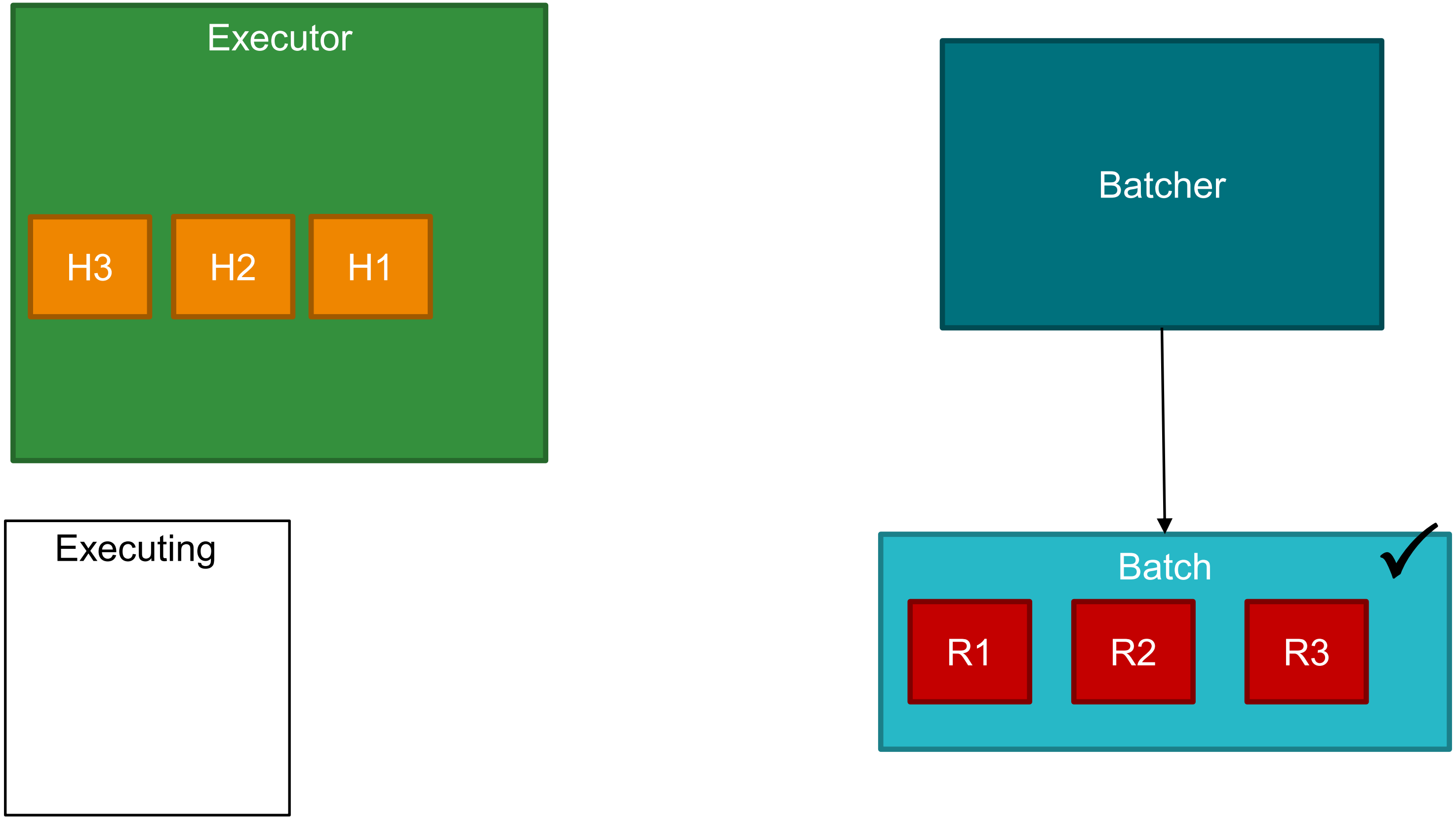
```
task sendEmail(const User& user) {
 const UserPrefs prefs = co_await getUserPrefs(user.id);




}
```

Implementing the Idea - Batcher

Let's create a wrapper type: `Batcher`

- intercepts the calls to `getUserPrefs(user.id)`
- stores the arguments
- executes the function
- resumes the coroutines

Implementing the Idea - Batcher



Implementing the Idea - Batcher

Needs to know the input and return type

```cpp
template<class T, class R>
struct Batcher {







};
```

Implementing the Idea - Batcher

Stores

- the executor

```
template<class T, class R>
struct Batcher {
  …
  Executor& ex_;




};
```

Implementing the Idea - Batcher

Stores

- the executor
- the function

```
template<class T, class R>
struct Batcher {
  …
  Executor& ex_;
  function<vector<R>(vector<T>)> op_;



};
```

Implementing the Idea - Batcher

Stores

- the executor
- the function and whether to execute it

```
template<class T, class R>
struct Batcher {
  …
  Executor& ex_;
  function<vector<R>(vector<T>)> op_;
  function<bool(const vector<T>&)> should_exec_;


};
```

Implementing the Idea - Batcher

Stores

- the executor
- the function and whether to execute it
- the current batch

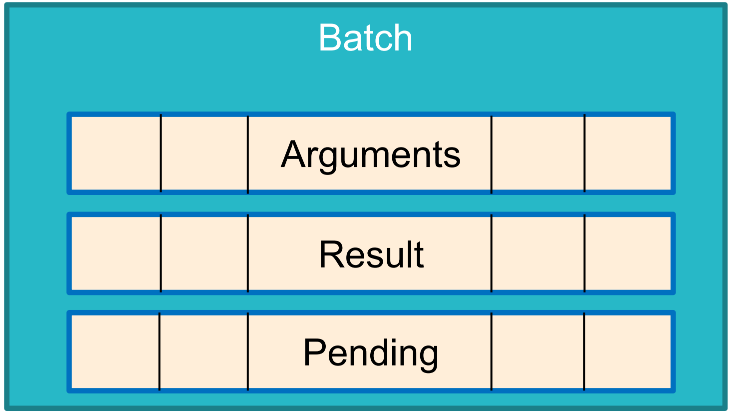
```
template<class T, class R>
struct Batcher {
  …
  Executor& ex_;
  function<vector<R>(vector<T>)> op_;
  function<bool(const vector<T>&)> should_exec_;
  struct Batch { … };
  shared_ptr<Batch> current_batch_;
};
```

Implementing the Idea - Batcher

Let’s look at the `Batch`

```
struct Batcher::Batch {



};
```

Implementing the Idea - Batcher



Implementing the Idea - Batcher

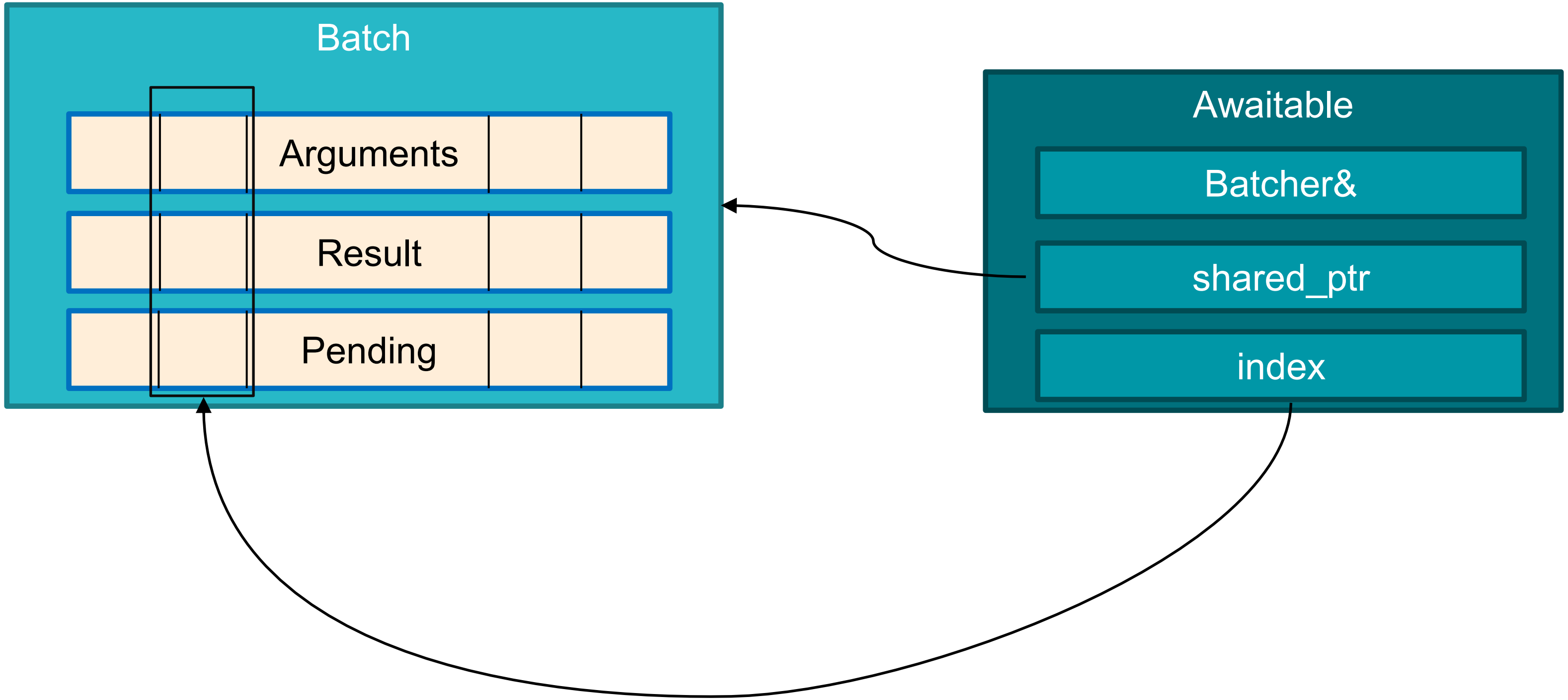
```
struct Batcher::Batch {
  vector<T> arguments;
  vector<R> result;
  vector<coroutine_handle<>> pending;
};
```

Implementing the Idea - Batcher

`Batcher` is callable
and returns an awaitable object

```
template<class T, class R>
struct Batcher {
  struct Awaitable { … };

  Awaitable operator()(T arg);
  …


};
```

Implementing the Idea - Batcher



Implementing the Idea - Batcher

```
struct Batcher::Awaitable {



  Batcher& batcher_;
  shared_ptr<Batch> batch_;
  size_t index_ = -1;
};
```

Implementing the Idea - Batcher

How is the `Awaitable` constructed?

```
template<class T, class R>
struct Batcher {

  Awaitable operator()(T arg) {
    current_batch_->args.push_back(arg);
    size_t index = current_batch_->args_.size() - 1;
    return Awaitable{*this, current_batch_, index};
  }
  …
};
```

Implementing the Idea - Batcher

`Awaitable` needs to implement the special methods

Let's do that!

Implementing the Idea - Batcher

Check if it’s ready

```
struct Batcher::Awaitable {
  bool await_ready() {
    return batcher_.maybe_execute();
  }




  …
};
```

Implementing the Idea - Batcher

Get the result once ready

```
struct Batcher::Awaitable {
  bool await_ready();
  R await_resume() {
    return batch_->results.at(index_);
  }



  …
};
```

Implementing the Idea - Batcher

Suspend

```
struct Batcher::Awaitable {
  bool await_ready();
  R await_resume();
  void await_suspend(coroutine_handle<> h) {
    batch_->pending_.push_back(h);
  }
  …
};
```

Implementing the Idea - Batcher

Suspend

```
struct Batcher::Awaitable {
  bool await_ready();
  R await_resume();
  coroutine_handle<> await_suspend(coroutine_handle<> h) {
    batch_->pending_.push_back(h);
    auto maybe_coro = batch_.executor_.pop_next_coro();
    return maybe_coro.value_or(std::noop_coroutine());
  }
  …
};
```

Implementing the Idea - Batcher

Only thing left: `maybe_execute()`

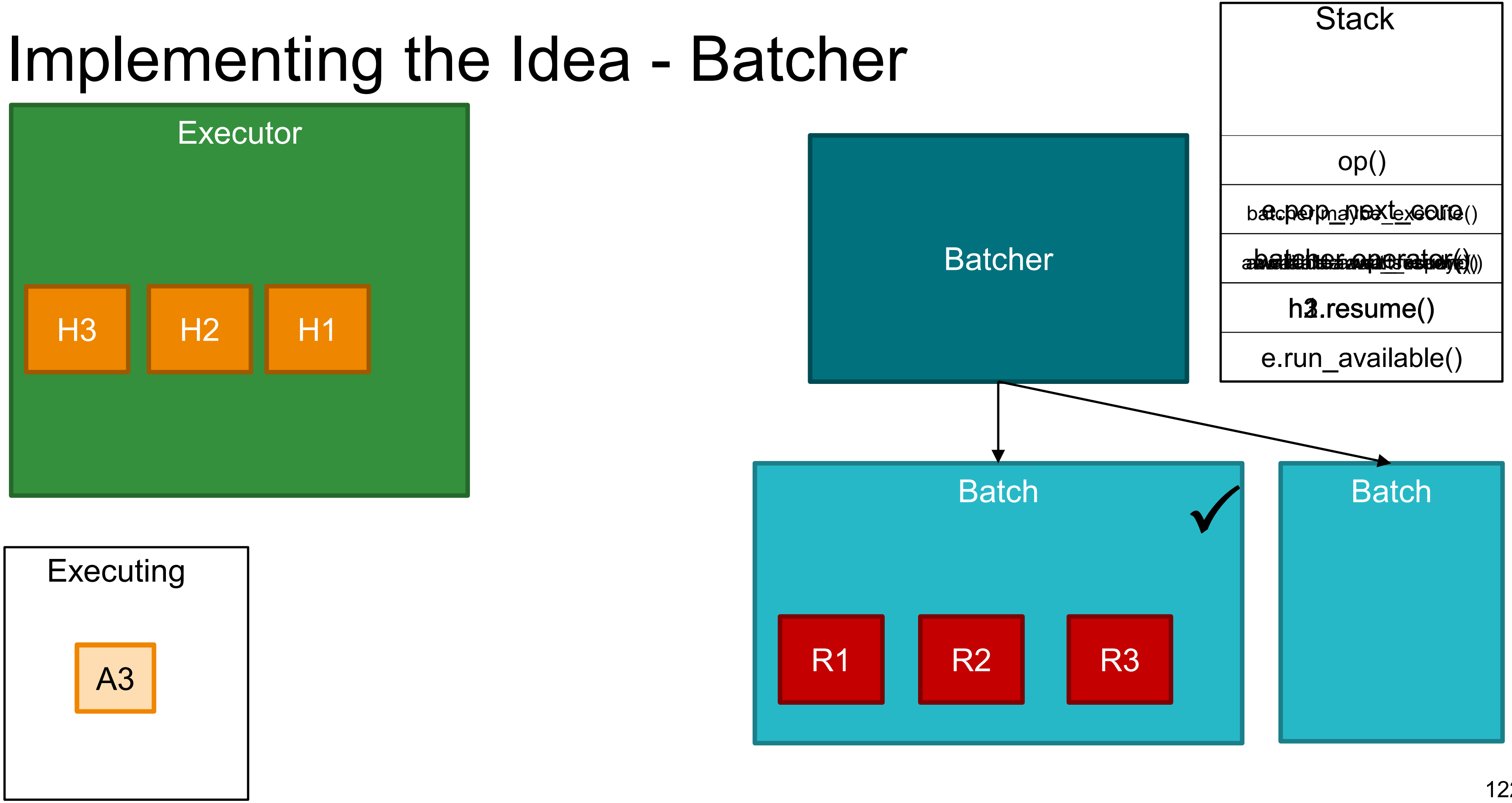
```
template<class T, class R>
struct Batcher {
  bool maybe_execute() {
    if( !should_exec_(batch_->args) ) return false;
    batch_->results = op_(batch_->args);
    executor_.submit(batch_->pending);
    batch_ = make_shared<Batch>();
    return true;
  }
};
```

Implementing the Idea - Batcher

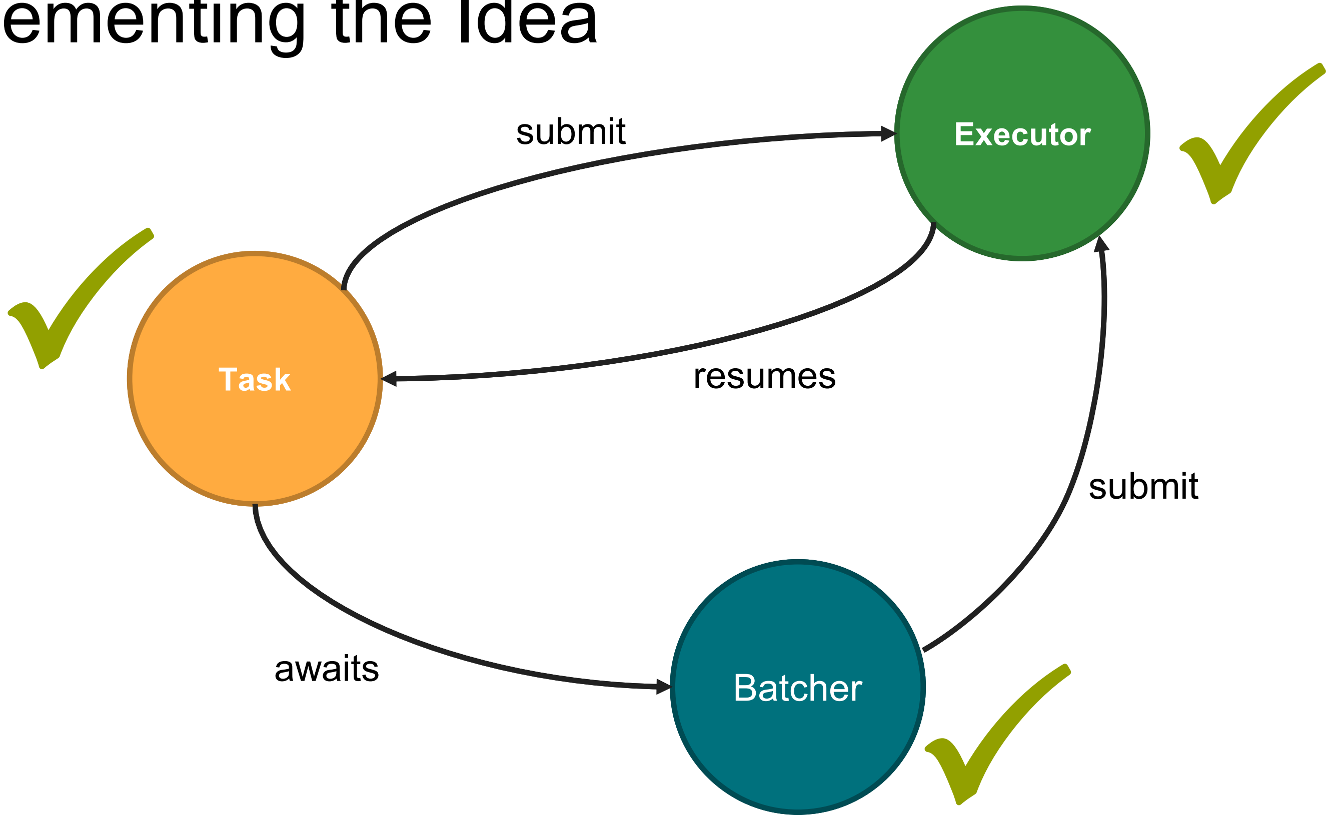
`Batcher` is complete!

```
Executor e;
Batcher<User, UserPref> getPrefs{e, getUserPrefs, size_eq(10)};

auto sendEmail = [&](const User& user) -> task {
 const UserPrefs prefs = co_await getPrefs(user.id);

};
```

Implementing the Idea - Batcher



Implementing the Idea



Questions?

Putting all together

Putting all together

```
Executor e;
Batcher<…> getUserPrefs{…};
Batcher<…> sendNotifs{…};

auto sendEmail = [&](const User& user) -> task {
 const UserPrefs prefs = co_await getUserPrefs(user.id);
 if (prefs.wantsEmailNotification) {
   co_await sendNotifs(prefs.notificationEmail);
 }
};

for(auto& user : users) {
  e.submit(sendEmail(user));
}

e.run_available();
```

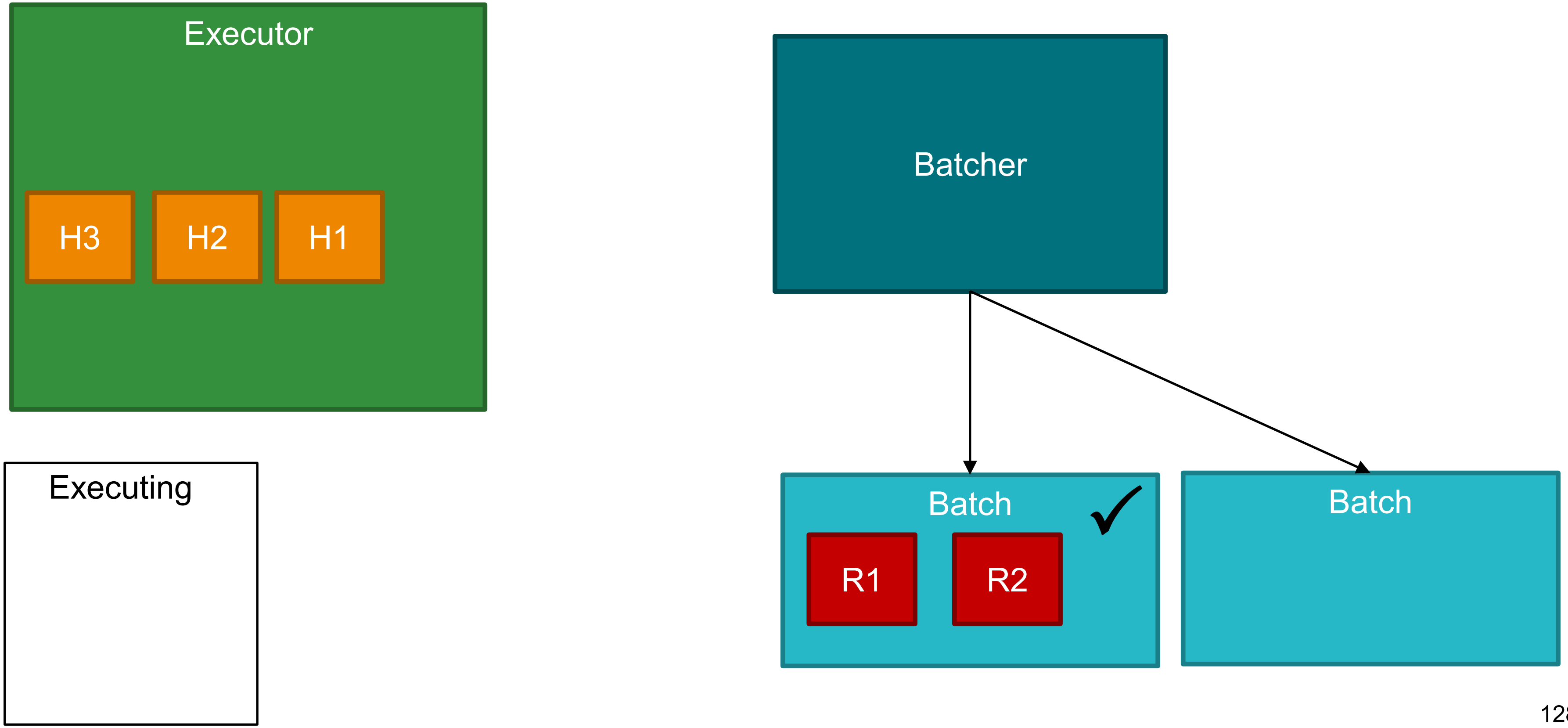
Problem

What if we finish the available coroutines before completing?

Example

- Batching with no limit
- Batching limit not a perfect divisor of the initial vector

Problem



Problem

Result

- the executor has no pending coroutines
- the batch has pending coroutines

Some tasks do not complete

Solution

Force the batch execution

```
template<class T, class R>
struct Batcher {
  bool maybe_execute(bool force = false) {
    bool exec = force || should_exec_(batch_->args);
    if( !exec ) return false;
    batch_->results = op_(batch_->args);
    executor_.submit(batch_->pending);
    batch_ = make_shared<Batch>();
    return true;
  }
};
```

Problem

When do we stop calling force?

Solution

When do we stop calling force?

When all tasks completed

Solution

Let’s keep track of the tasks

```
struct Executor {
  void submit(task t) {
    counter_++;
    auto handle = move(t).release();
    handle.promise().counter_ = &counter_;
    pending_.push_back(handle);
  }

  size_t counter_ = 0;
};
```

Solution

Decrease the counter when a task completes

```
struct task::promise_type {
  void return_void() {
    *counter -= 1;
  }
  size_t* counter_;
};
```

Solution

Return whether all tasks completed

```
struct Executor {
  bool run_available() {
    for(auto next = pop_next_coro(); next; next = pop_next_coro()) {
      next.resume();
    }
    return counter_ == 0;
  }
};
```

Solution

```
Executor e;
Batcher<…> getUserPrefs{…};
Batcher<…> sendNotifs{…};

auto sendEmail = [&](const User& user) -> task {…};

for(auto& user : users) {
  e.submit(sendEmail(user));
}

while(e.run_available()) {
  getUserPrefs.maybe_execute(true);
  sendNotifs.maybe_execute(true);
}
```

Problem

We are calling `maybe_execute()` on every batcher

It’s unnecessary

Solution

Execute only the first `maybe_execute()`

Solution

Skip the execution if no args

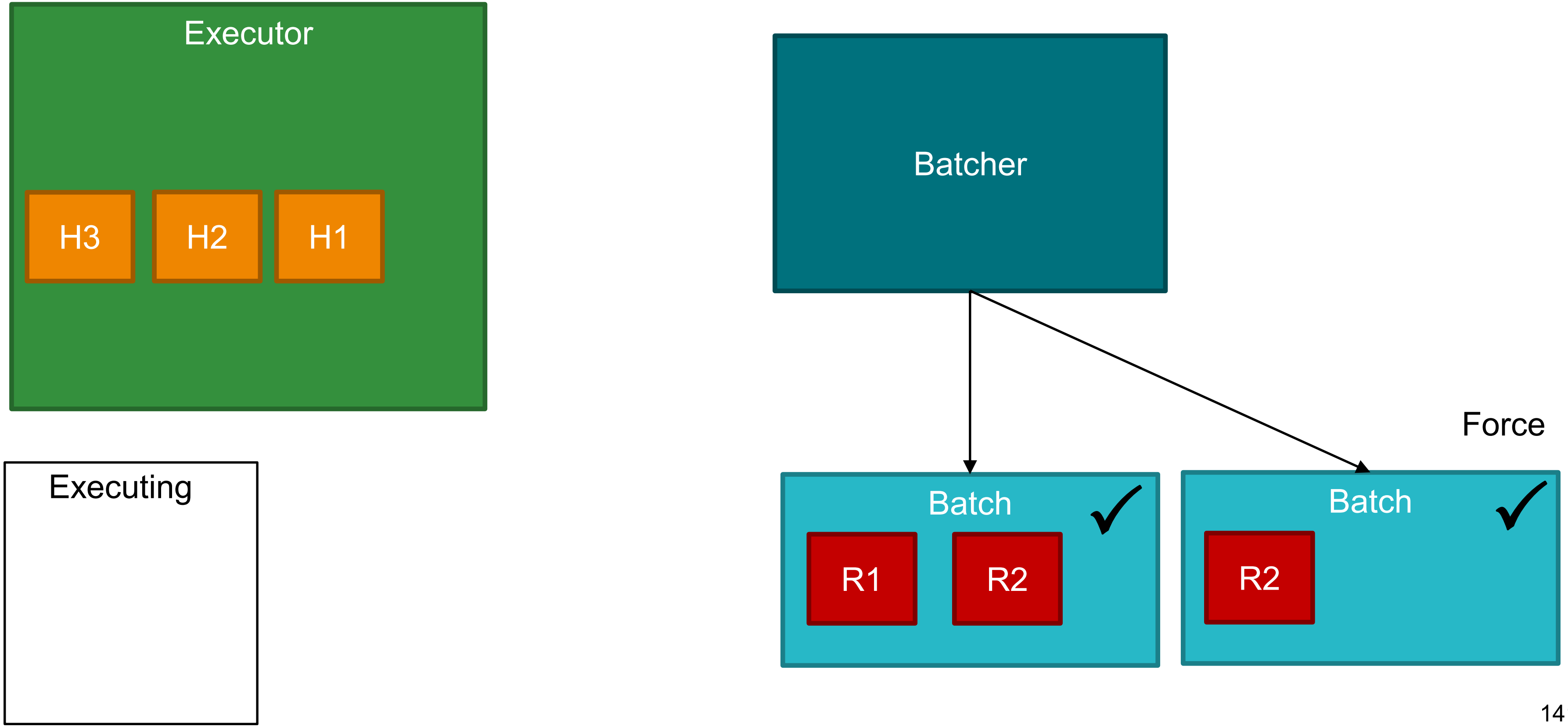
```
template<class T, class R>
struct Batcher {
  bool maybe_execute(bool force = false) {
    bool exec = force || should_exec_(batch_->args);
    if( !exec || batch_->args.empty() ) return false;
    batch_->results = op_(batch_->args);
    executor_.submit(batch_->pending);
    batch_ = make_shared<Batch>();
    return true;
  }
};
```

Solution

Execute only the first `Batcher`

```
…
while(e.run_available()) {
  getUserPrefs.maybe_execute(true) ||
    sendNotifs.maybe_execute(true);
}
```

Problem


Executor
H3
H2
H1
Executing
Batcher
Force
Batch
R1
R2
Batch
R2

Final Solution

```
Executor e;
Batcher<…> getUserPrefs{…};
Batcher<…> sendNotifs{…};

auto sendEmail = [&](const User& user) -> task {
 const UserPrefs prefs = co_await getUserPrefs(user.id);
 if (prefs.wantsEmailNotification) {
   co_await sendNotifs(prefs.notificationEmail);
 }
};

for(auto& user : users) { e.submit(sendEmail(user)); }

run_to_completion(e, getUserPrefs, sendNotifs);
```

Questions?

Benchmark

Benchmark

We implemented the `task`, an `executor` and a `Batcher` with coroutines
We can now automatically batch our operations

How does it compare to manual batching?

Manual Batching

```
for(const User& user : users) {
  const UserPrefs prefs =
    getUserPrefs({user.id}).at(0);
  if (prefs.wantsEmailNotification) {
    sendNotifs({prefs.notificationEmail});
  }
}
```

```
vector<UserId> userIds;
transform(users.begin(), users.end(),
  back_inserter(userIds),
  [](auto& user) { return user.id; });

vector<UserPrefs> preferences =
  getUserPrefs(userIds);
remove_if(preferences.begin(),
  preference.end(),
  [](auto& pref) {
    return !pref.wantsEmailNotification;});

vector<EmailAddress> emails;
transform(preferences.begin(),
  preference.end(),
  back_inserter(emails),
  [](auto& pref) {
    return pref.notificationEmail; });

sendNotification(emails);
```

Coro Batching

```
for(const User& user : users) {
  const UserPrefs prefs =
    getUserPrefs({user.id}).at(0);
  if (prefs.wantsEmailNotification) {
    sendNotifs({prefs.notificationEmail});
  }
}
```

```
Executor e;
Batcher<…> getUserPrefs{…};
Batcher<…> sendNotifs{…};

auto sendEmail = [&](const User& user)->task {
  const UserPrefs prefs =
    co_await getUserPrefs(user.id);
  if (prefs.wantsEmailNotification) {
    co_await sendNotifs(
      prefs.notificationEmail);
  }
};

for(auto& user : users) {
  e.submit(sendEmail(user));
}

run_to_completion(e,getUserPrefs,sendNotifs);
```

Benchmark

Is performance comparable?

Simulation

- 10 ms delay per call
- 50 us delay per item processed (100 users, 50 email sent)

Test	Time (us)	Iterations	Baseline
No Batching	1’552’542	13	49.99
Manual Batching	31’055	492	-
Coro Batching	27’748	564	0.89

Benchmark

Overhead?

Simulation without delays

Test	Time (ns)	Iterations	Baseline
No Batching	17’004	446’422	1.61
Manual Batching	10’586	655’351	-
Coro Batching	30’490	224’300	2.88

Benchmark

Overhead?

Simulation without delays

Test	Time (ns)	Iterations	Baseline
No Batching	17’004	446’422	1.61
Manual Batching	10’586	655’351	-
Coro Batching	30’490	224’300	2.88
Coro Batching (Opt)	12’546	548’914	1.18

Conclusion

Conclusion

A new point on the performance ⟷ readability trade-off

Learned

- Main customization points of coroutines
- Coroutines can be used for more than just async programming
- Coroutines are powerful, but at times complicated

Conclusion

Find the full implementation at

github.com/MakersF/cppcon-2021-corobatch

Library exploring the concept

github.com/MakersF/corobatch

Thank you!

Useful Material

- Lewis Baker – lewissbaker.github.io/
- Dawid Pilarski – blog.panicsoftware.com
- Gor Nishanov – “Nano-coroutines to the Rescue!”