# Coroutine Patterns and How to Use Them:

## Problems and Solutions Using Coroutines In a Modern Codebase

FRANCESCO ZOFFOLI

Cppcon
The C++ Conference

2023 | October 01 - 06

# About Me

- Software engineer building monitoring systems at Meta
- Passionate about C++
- Author of the book “C++ Fundamentals” – Packt
- I like writing and talking about C++

makers.f.dev@gmail.com

# Outline

- Motivation
- Overview
- Patterns
    - Lifetime
    - Exceptions
    - RAII
    - Synchronization
- Conclusions

# Motivation

- 40 years of
  - Experience
  - Accumulated knowledge

# Motivation

Coroutines introduce new paradigm

# Motivation

- Build shared knowledge

# Overview

- Folly - github.com/facebook/folly
- Most libraries are similar

# Overview

Key concepts

- Task
- Executor

# Overview – Task

- Owns the coroutine state
- Lazy
  - Convertible to eager
- Propagates executor
- Sticky to executor

# Overview – Task - Lazy

```
Task<> foo() {
  println("Hello");
  co_await sleep(1);
}

auto t = foo();
println("world");
co_await move(t);
```

# Overview – Executor

- Executes synchronous functions
- Could be multi-threaded
- Coroutines are split in synchronous functions

# Overview – Executor

```
Task<> foo() {
  int a = 42;
  a -= 10;
  int bytes = co_await send(a);
  if (bytes == -1 ) {
    handle_error();
  }
}
```

# Overview – Executor

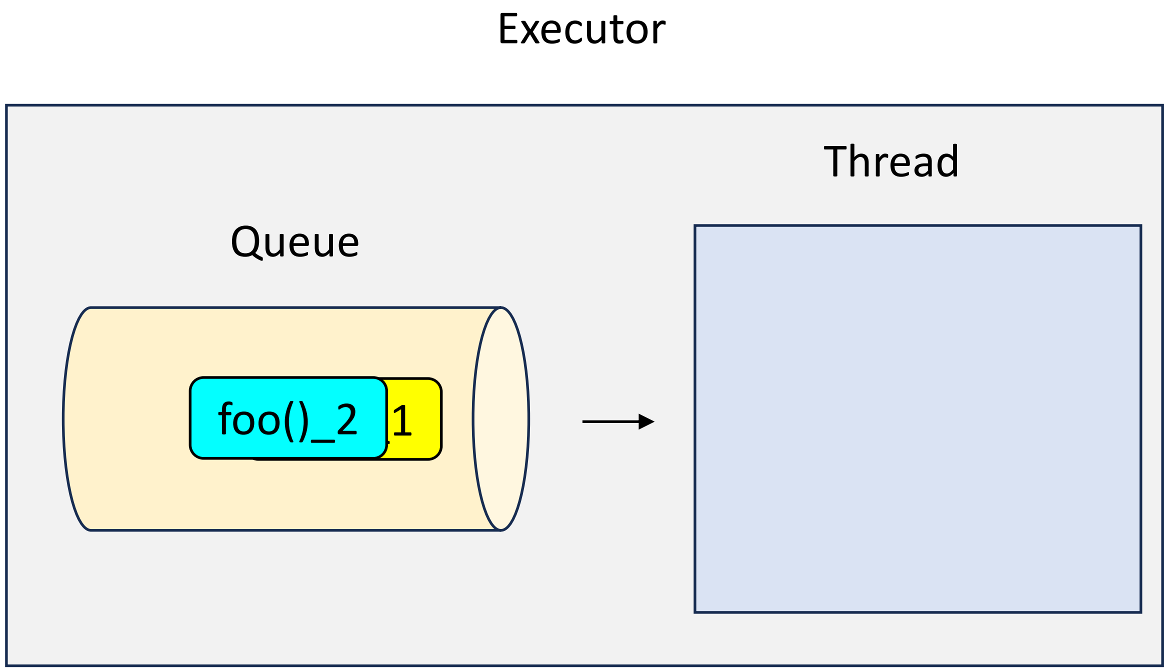
```
Task<> foo() {
   int a = 42;
   a -= 10;
   int bytes = co_await send(a);
   if (bytes == -1 ) {
      handle_error();
   }
}
```

Executes until continuation

co_await

Executes after continuation

# Overview - Executor



# Patterns

- Lifetime
- Exceptions
- RAII
- Synchronization

# Lifetime

# Lifetime

- Lots of worries
- Use Structured Concurrency[1]
- ASAN works great

[1] https://ericniebler.com/2020/11/08/structured-concurrency/

# Lifetime

Member coroutines implicitly capture `this`

# Lifetime

Let’s play a game

# Lifetime

```
struct Bar {
  int data = 0;

 Task<int> foo() {
    co_await sleep(1);
    co_return data;
  }
};
```

```
Task<int> mul_2(Task<int> t){
  int val = co_await move(t);
  co_return 2 * val;
}
```

# Lifetime

Let’s play a game

# Lifetime

```
std::vector<int> vec;
vec.front() += 1;
```

# Lifetime

```
std::vector<int> vec;
vec.clear();
```

# Lifetime

```
Bar a;
co_await a.foo();
```

```
Bar a;
Task<int> t = a.foo();
co_await move(t);
```

5

# Lifetime

```
Task<int> t = Bar{}.foo();
co_await move(t);


co_await Bar{}.f();
```

5

# Lifetime

```
Bar a;
Task<int> t = mul_2(a.foo());
co_await move(t);

Task<int> t = mul_2(Bar{}.foo());
co_await move(t);

co_await mul_2(A{}.foo());
```

5

# Lifetime – Shared knowledge

- Keep object alive until all methods have completed

# Lifetime

When does this concretely happens?

# Lifetime

When does this concretely happens?

In loops!

```
vector<Task<>> tasks;
for(int index = …) {
  Foo foo{index};
  tasks.push_back(foo.bar());
}
co_await collectAll(tasks);
```

# Lifetime

When does this concretely happens?

Lambdas!!

```
Task<int> d = mul_2([=]() { co_return i; }());
co_await move(d);
```

# Lifetime

When does this concretely happens?

Lambdas!!

```
co_await mul_2([=]() { co_return i; }());

auto lam = [=]() { co_return i; };
co_await mul_2(lam());
```

# Lifetime

Use co_invoke!

```
Task<int> d = mul_2(co_invoke([=]() { co_return i; }));
co_await move(d);
```

# Lifetime – Shared knowledge

- Keep object alive until all methods have completed
- Ensure lambda objects are kept alive (using `co_invoke`)

# Lifetime

Nothing new!

Think: task reference this pointer

```
vector<tuple<int&>> vec;
for(int index = …) {
  vec.emplace_back(index);
}
```

Who wouldn’t spot this?

# Lifetime

Scheduling in the background

# Lifetime

```
{
  Foo a;
  startInBackground(a.bar());
  co_await do_other_stuff();
}
```

Big no no!

Need to join background work

# Lifetime

Wait for all work started in the same scope

```
{
  Foo a;
  auto bgWork = startInBackground(a.bar());
  co_await do_other_stuff();
  co_await move(bgWork);
}
```

# Lifetime

Waiting for many background Tasks – AsyncScope

```
Foo a;
AsyncScope s;
for(int index = …) {
  co_await s.schedule(a.bar(index));
}
co_await do_other_stuff();
co_await s.join();
```

# Lifetime - Shared knowledge

- Keep object alive until all methods have completed
- Ensure lambda objects are kept alive (using `co_invoke`)
- Always join work before leaving the scope

# Lifetime

Anyone sees a problem with the code above

What if `do_other_stuff()` throws?

Still need to join!

# Exceptions

# Exceptions

Await in catch

```
try {
  co_await do_other_stuff();
} catch (…) {
  co_await asyncScope.join();
}
```

Compiler error!

Not allowed to co_await inside a catch block

# Exceptions

## Capture the exception

```
exception_ptr eptr;
try {
  co_await do_other_stuff();
} catch (…) {
  eptr = current_exception();
}
if (eptr) {
  co_await asyncScope.join();
  rethrow_exception(eptr);
}
```

# Exceptions

Capture the exception (2)

```
auto res = co_await co_awaitTry(do_other_stuff());
if (res.hasException()) {
   co_await asyncScope.join();
}
// Rethrows if it doesn’t have a value
res.value();
```

# Exceptions - Shared knowledge

- To do async work in catch block, capture the exception in a wrapper
  - Use libraries to help

# RAII

Different solutions for

- Class hierarchies and members
- Automatic variables

# RAII – Classes

Async cleanup pattern

- Define async cleanup method
- Parents/owners call the method, recursively

# RAII – Classes

```
class Foo {
  Task<> cleanup() {
    co_await collectAll(as.join(), m1.cleanup());
  }

  Bar m1;
  AsyncScope as;
};
```

# RAII – Classes

- cleanup() shouldn't throw
- Tip: assert in the destructor that cleanup() was called

A bit of manual work ☹

# RAII – Scopes

Need to ensure `cleanup()` is called

# RAII – Scopes

- Avoid exceptions
- Testing

# RAII - Shared Knowledge

- Use cleanup pattern to join async work in classes

# RAII

“This is bad! I’ll use blockingWait() in the destructor”

```
~MyClass() {
   blockingWait(collectAll(as.join(), m1.cleanup()));
};
```

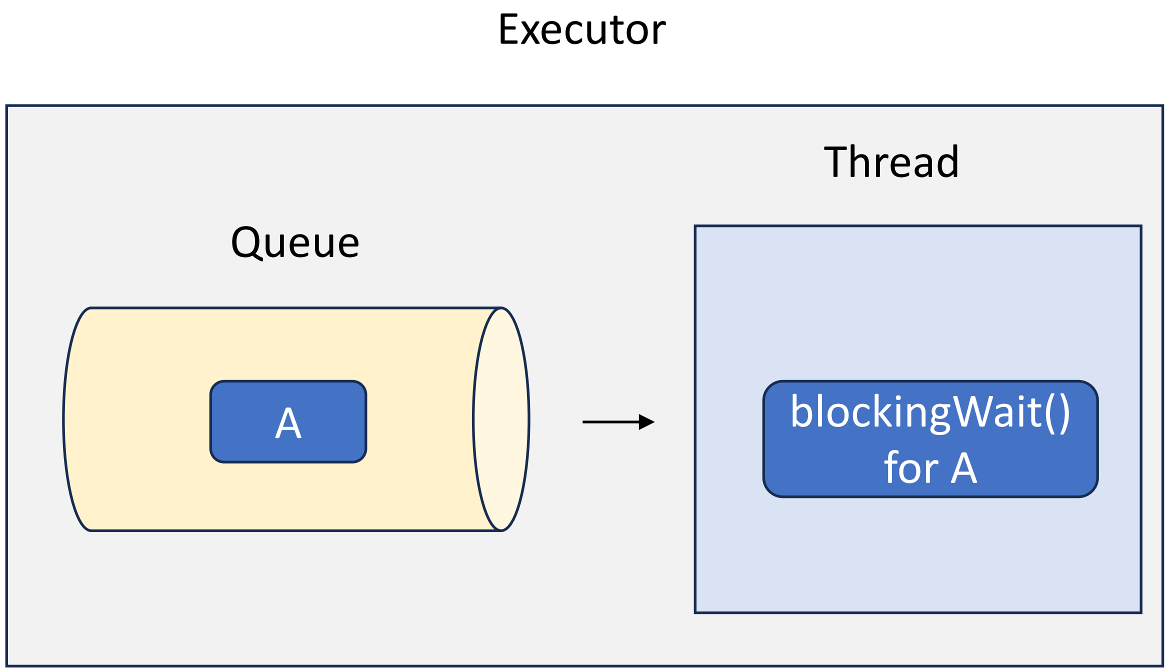
# RAII

DON'T!!

# RAII

Why?

1. blockingWait() blocks a thread
2. continuation is scheduled
3. no thread available to run it



# RAII

```
struct Foo {
  Task<> do_work() {
    co_await
      as.schedule(sleep(1s));
  }

  Task<> cleanup() {
    co_await as.joinAsync();
  }
  ~Foo() {
    blockingWait(cleanup());
  }

  AsyncScope as;
};
```

# RAII – Shared Knowledge

- Use cleanup pattern to join async work in classes
- Don't use `blockingWait()` in a destructor

# Synchronization

# Synchronization

Needed?

# Synchronization

No shared access? → No

```
int a = 10;
co_await reschedule_on_current_executor;
// Resume in different thread
do_stuff(a);
```

# Synchronization

Shared Access? Depends

- ST executor →No
- MT executor → Yes

```
int a = 10;
co_await collectAll(
  increaseBy(a, 1),
  increaseBy(a, 2));
```

# Synchronization

Which tools?

- Regular mutexes
- Coro mutex

# Synchronization – Regular

DON’T HOLD ACROSS SUSPENSION POINTS!!!

- UB if held across suspension points
- Only for small sections
- Useful for advanced features/non-coro code

# Synchronization – Regular

Use lambda format

```
synchronized.withWLock([&](auto&) {
    …
  });
```

- co_await → compile error
- guard patter is risky

# Synchronization – Shared Knowledge

- Do not hold regular lock across suspension points

# Synchronization – Coro

Support RAII!

# Synchronization – Coro

Access mutex protected data in destructor

- try_lock()

# Synchronization – Shared Knowledge

- Do not hold regular lock across suspension points
- Use coro mutexes, including guards
- Use `try_lock()` to for coro mutexes in destructor

# Conclusions

- Saw patterns to do and avoid
- Build shared knowledge
- Invite people to share experiences
- Identify unsolved needs

# Thank you