Coroutines

All you need to know about the coroutines

Dawid Pilarski

dawid.pilarski@tomtom.com
blog.panicsoftware.com

Agenda

Coroutine theory - what are the coroutines?

Practical part I - using cppcoro

Theory - implementing own coroutines types

Practical part II - implementing own coroutines types

Coroutine theory - what are the coroutines?

What are the coroutines?

Coroutines are generalization of the function, that can be:

- created

What are the coroutines?

Coroutines are generalization of the function, that can be:

- created
- called

What are the coroutines?

Coroutines are generalization of the function, that can be:

- created
- called
- returned from

What are the coroutines?

Coroutines are generalization of the function, that can be:

- created
- called
- returned from
- suspended

What are the coroutines?

Coroutines are generalization of the function, that can be:

- created
- called
- returned from
- suspended
- resumed

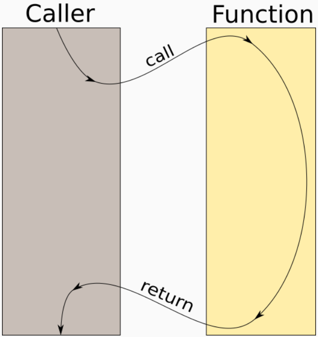
What are the coroutines?

Coroutines are generalization of the function, that can be:

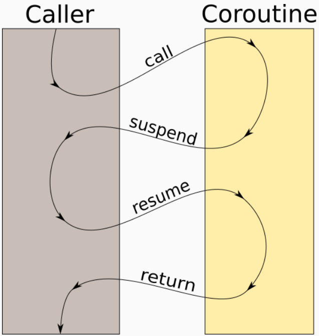
- created
- called
- returned from
- suspended
- resumed
- destroyed

Coroutine flowchart

Function's flow:

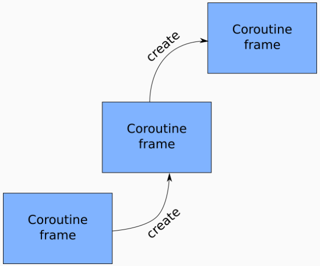


Coroutine flow:

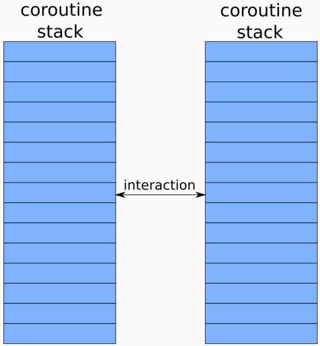


Possible coroutines implementations

Language based



Library based



Closer look into Boost.Fiber

- Need to allocate the stack for the Fiber/Coroutine

Closer look into Boost.Fiber

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below

Closer look into Boost.Fiber

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance

Closer look into Boost.Fiber

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- Hard to optimize by compilers

Closer look into built-in coroutines

include in the view also boost.fiber for comparison

- Need to allocate the frame for the Coroutine

Closer look into built-in coroutines

include in the view also boost.fiber for comparison

- Need to allocate the frame for the Coroutine
- Can be suspended only from the top level functions

Closer look into built-in coroutines

include in the view also boost.fiber for comparison

- Need to allocate the frame for the Coroutine
- Can be suspended only from the top level functions
- Minimal memory allocation

Closer look into built-in coroutines

include in the view also boost.fiber for comparison

- Need to allocate the frame for the Coroutine
- Can be suspended only from the top level functions
- Minimal memory allocation
- Easy to optimize by compilers

Coroutine declarations

Same as functions

```
// returned-type   name      arguments
//|-------------| |-------| |--------------|
   generator<int> fibonacci (int from_value);
```

Whether the function is a coroutine depends on it’s definition.

3 new keywords

`co_return`
: Returning (or not) value and finishing the coroutine

`co_yield`
: Returning intermediate value from the coroutine

`co_await`
: Awaiting completion of the "task"

Practical part I - using cppcoro

generators - Fibonacci sequence

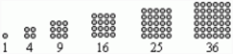
```
cppcoro::generator<unsigned long long> fibonacci_gen() {
  std::array arr{0ull, 0ull};
  unsigned long long result=0;

  do {
    co_yield result;
    if(result == 0 and arr == std::array{0ull, 0ull})
      result = 1;
    else if (result == 1 and arr == std::array{0ull, 0ull})
      arr = {0, 1};
    else{
      arr[0] = arr[1];
      arr[1] = result;
      result = arr[0] + arr[1];
    }
  } while (result >= arr[1]);
}
```

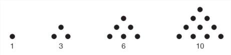
generators - excercise

Implement any generator:

- square number series



- triangular number series



tasks and events

```
void test(){
  single_consumer_event event;
  cppcoro::sync_wait(cppcoro::when_all_ready(
    [&]() -> cppcoro::task<> {
      while(true){
        co_await event;
        event.reset();
        std::this_thread_sleep(500ms)
      }
    }(),
    [&]() -> cppcoro::task<>{
      while(true){
        event.set();
      }
    }()
  ));
}
```

tasks and events excercise

Write an application, that will at the same time (almost)

CHECK IF IT WORKS

implement framework for that.

- Read large content from a file
- Display dots every second

other (for now only msvc)

- mutexes
- file I/O operations
- networking operations

other (for now only msvc)

```cpp
cppcoro::task<int> count_lines(std::string path)
{
  auto file = co_await cppcoro::read_only_file::open(path);

  int lineCount = 0;
  char buffer[1024];
  size_t bytesRead;
  std::uint64_t offset = 0;
  do
  {
    bytesRead = co_await file.read(offset, buffer, sizeof(buffer));
    lineCount += std::count(buffer, buffer + bytesRead, '\n');
    offset += bytesRead;
  } while (bytesRead > 0);

  co_return lineCount;
}
```

Theory - implementing own coroutines types

promise_type

- promise_type is responsible for coroutine's behavior:

promise_type

- promise_type is responsible for coroutine's behavior:
    - on coroutine's start

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start
  - on throwing unhandled exception

promise_type

- promise_type is responsible for coroutine's behavior:
    - on coroutine's start
    - on throwing unhandled exception
    - on coroutine's end

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start
  - on throwing unhandled exception
  - on coroutine's end
  - on returning the value

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start
  - on throwing unhandled exception
  - on coroutine's end
  - on returning the value
  - on yielding the value

promise_type

- promise_type is responsible for coroutine's behavior:
    - on coroutine's start
    - on throwing unhandled exception
    - on coroutine's end
    - on returning the value
    - on yielding the value
    - partially on waiting for the task's completion

promise_type

- promise_type is responsible for coroutine's behavior:
    - on coroutine's start
    - on throwing unhandled exception
    - on coroutine's end
    - on returning the value
    - on yielding the value
    - partially on waiting for the task's completion
- promise_type is strongly connected with coroutine's returned type

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start
  - on throwing unhandled exception
  - on coroutine's end
  - on returning the value
  - on yielding the value
  - partially on waiting for the task's completion
- promise_type is strongly connected with coroutine's returned type
  - it can be a member of the returned type
    `returned_type::promise_type`

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start
  - on throwing unhandled exception
  - on coroutine's end
  - on returning the value
  - on yielding the value
  - partially on waiting for the task's completion
- promise_type is strongly connected with coroutine's returned type
  - it can be a member of the returned type
    `returned_type::promise_type`
  - it can be defined as (CHECK IT)
    `std::coroutine_traits<returned_type>::promise_type`

coroutine body

ADD highlighting to the presentation

Each time we write coroutine, compiler modifies it's body into following form:

```
{
  promise_type promise;
  auto return_object = promise.get_return_object();
  co_await promise.initial_suspend();

  try{
    //our coroutine_body
  }catch(...) {
    promise.unhandled_exception();
  }

  final_suspend:
  co_await promise.final_suspend();

  return return_object;
}
```

promise_type extensions

We can extend the coroutine body with:

- (mandatory) support for returning void or returning value
- custom memory allocation algorithm (custom `operator new`)
- (optional) support for yielding intermediate values

co_return - support for returning from coroutine

`co_return` is a new keyword

Usage:

without expression:

```
co_return
```

with void expression:

```
co_return <void expression>
```

Translated to:

```
<expression>;
promise.return_void();
goto final_suspend;
```

co_return - support for returning from coroutine

`co_return` is a new keyword

Usage:

Translated to:

with non-void expression:

```
co_return <expression>
```

```
promise.return_value(<expression>);
goto final_suspend;
```

co_yield - support for returning intemediate values

co_yield is a new keyword

Usage:

```
co_yield <non-void expression>
```

Translated to:

```
co_await promise.
        yield_value(<expression>)
```

co_await shortly

`co_await std::experimental::suspend_always{}` -
suspends the coroutine

`co_await std::experimental::suspend_never{}` -
does nothing

excercise here!

co_await

co_await is a new keyword

- represents awaiting for operations' completion
- it's argument is (usually) called awaitable
- it's result is usually called awaiter

co_await translation

If compiler meets the co_await it gets translated into following code:

```
{
  std::exception_ptr exception = nullptr;
  if (not a.await_ready()) {
    suspend_coroutine();

    <await_suspend>
  }

  resume_point:
  if(exception)
    std::rethrow_exception(exception);
  "return" a.await_resume();
}
```

Await suspend

await suspend is of the following form:

```
promise.await_suspend(this_coroutine_handle);
```

await_suspend might return following types:

- void

```
//if await_suspend returns void
try {
  a.await_suspend(coroutine_handle);
  return_to_the_caller();
} catch (...) {
  exception = std::current_exception();
  goto resume_point;
}
//endif
```

Await suspend

await suspend is of the following form:

```
promise.await_suspend(this_coroutine_handle);
```

await_suspend might return following types:

- void
- bool

```
//if await_suspend returns bool
bool await_suspend_result;
try {
  await_suspend_result = a.await_suspend(
                           coroutine_handle);
} catch (...) {
  exception = std::current_exception();
  goto resume_point;
}
if (not await_suspend_result)
  goto resume_point;
return_to_the_caller();
//endif
```

Await suspend

await suspend is of the following form:

```
promise.await_suspend(this_coroutine_handle);
```

await_suspend might return following types:

- void
- bool
- coroutine_handle

```
//if await_suspend returns coroutine_handle
decltype(a.await_suspend(
  std::declval<coro_handle_t>()))
another_coro_handle;
try {
  another_coro_handle = a.await_suspend(
                            coroutine_handle);
} catch (...) {
  exception = std::current_exception();
  goto resume_point;
}
another_coro_handle.resume();
return_to_the_caller();
//endif
```

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform

is performed only if promise has await_transform function declared

```
co_await promise.await_transform(<expr>);
```

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform
- acquiring awaiter

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform
- acquiring awaiter
    - co_await operator

is performed only if awaitable has co_await operator

```
auto&& awaiter =
       <awaitable>.operator co_await();
```

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform
- acquiring awaiter
    - co_await operator
    - global co_await operator

is performed only if awaitable there is matching global co_await operator

```
auto&& awaiter =
      operator co_await(<awaitable>);
```

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform
- acquiring awaiter
  - co_await operator
  - global co_await operator
  - awaitable to awaiter

```
auto&& awaiter = <awaitable>;
```

Practical part II - implementing own coroutines types

lazy

type for lazy initialization

requirements:

- synchronous (no support for multithreading + no support for co_await).
- no sharing of the value (no copy, only move constructor).
- interface simillar to the std::optional.

generator

type for generating sequences

requirements:

- synchronous (no support for multithreading + no support for co_await).
- next method should return the value and resume the coroutine.
- interface simillar to the std::optional

task

type for asynchronous operations

requirements:

- single-threaded
- asynchronous (support for the co_await)
- coroutine after finishing must resume the co_awaiting coroutine
- some kind of the executor needed to start the coroutines (GitHub)

event

type for communication of the tasks

requirements:

- stores information whether the event is set.
- stores the continuation object.
- launches the continuation on setting up the event.