C++ - coroutines

David Bednárek
Jakub Yaghob
Filip Zavoral

References

- https://www.youtube.com/watch?v=RhXaKOe3JZM
- https://blog.panicsoftware.com/coroutines-introduction/
- https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html

What are coroutines?

- Like a subroutines
  - Can be called
  - Can return when completed
- But with some differences
  - Can suspend themselves
  - Can be resumed (by someone else)

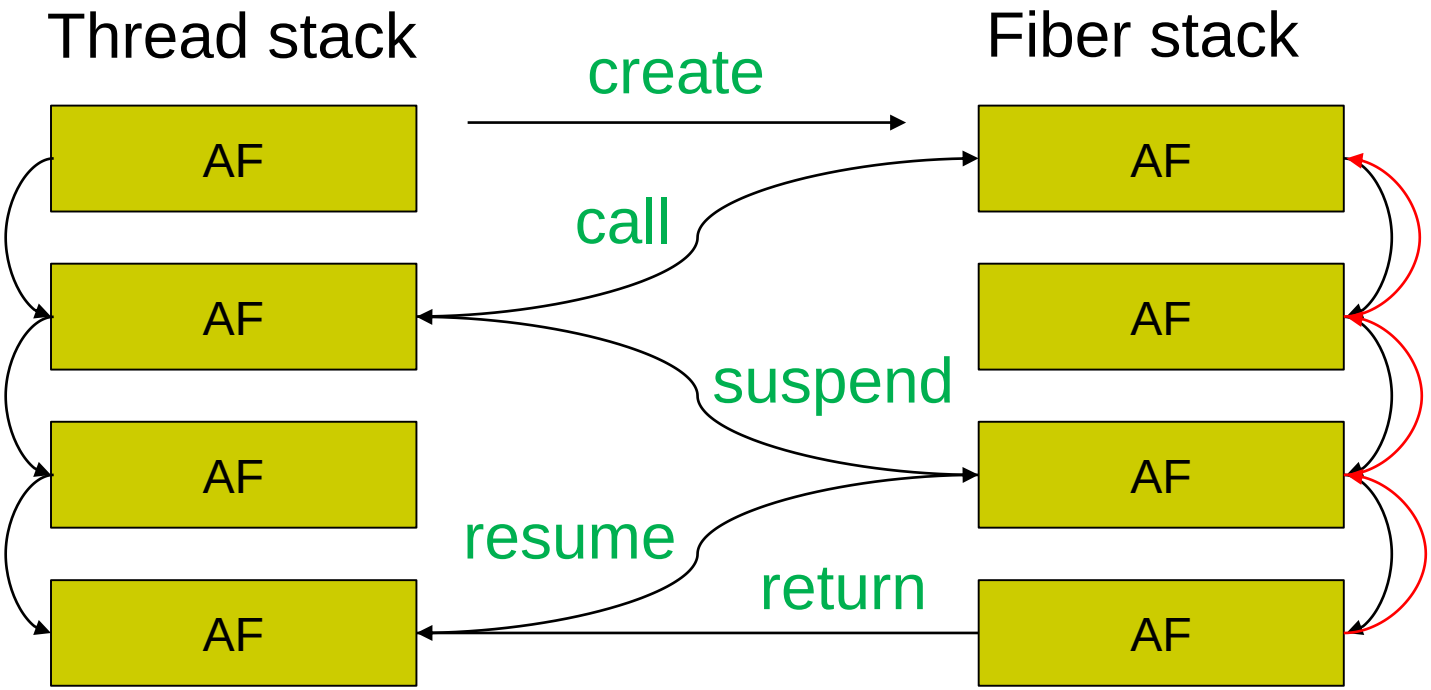
Why do we want coroutines?

- Event driven architectures
  - Asynchronous I/O
  - User interfaces
  - Simulations
- Generators
- Lazy evaluation
- Cooperative multitasking
  - Cheaper context-switch compared with threads

Stackful coroutines

- Stackful
  - Fibers, green threads, etc.
  - They have their own call stack
  - Their lifetime is independent to the caller code
  - Can be attached and detached to/from threads
  - Cooperative scheduling
  - Can be implemented as a library, no need for language support

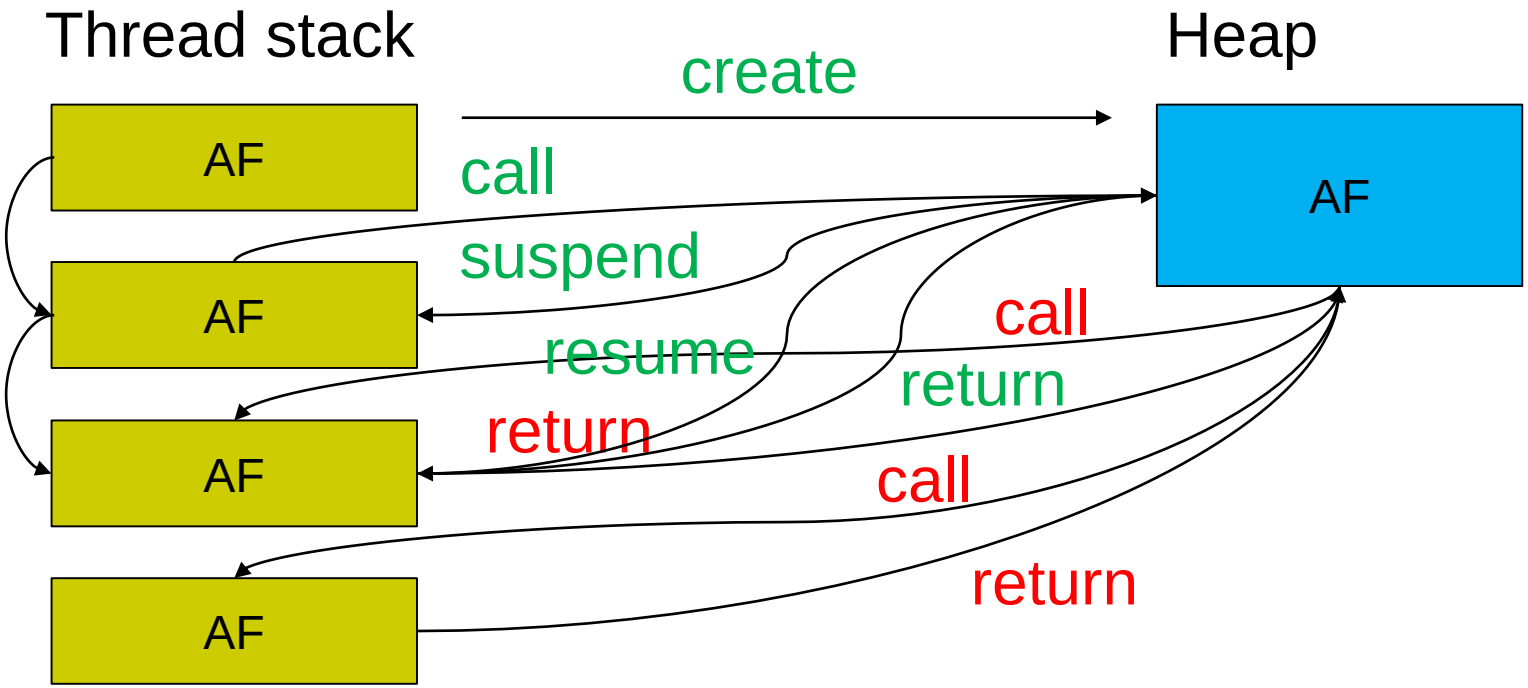
Stackful coroutines



Stackless coroutines

- Stackless
  - Use caller’s stack
  - Can be suspended only from the top level function
    - All function calls made by coroutine must return before suspend
  - Coroutine state saved on the heap
  - Require language level support
  - Usually lighter
  - C++ 20

Stackless coroutines



C++20 coroutines

- Stackless
- No higher level capabilities
  - Generators, resumable functions, and other predefined patterns
    - C#, JavaScript, Python, …
- Higher level capabilities will be added in the next C++ release

C++20 coroutines

- How to detect a coroutine?
  - Any use of coroutine keyword transforms a function to the coroutine
    - Expressions `co_await`, `co_yield`
    - Statement `co_return`

What does co_await?

- All local variables in the current function are saved to a heap allocated object
- Creates a callable object that, when invoked, will resume execution of the coroutine at the point immediately following evaluation of the `co_await` expression
- Calls (jumps to) a method of `co_await`'s target object a, passing that method the callable object from 2nd step

Coroutine handles

- Coroutine handle
  - Like a C pointer
  - Type `std::coroutine_handle<>`
  - Call `coroutine_handle::destroy` to avoid leaking memory, it destroys the state
  - Once destroyed, invoking coroutine handle has undefined behavior
  - Coroutine handle is valid for the entire execution of a coroutine , even as control flows in and out of the coroutine

What does co_await again?

- What does co_await a;
  - The compiler creates a coroutine handle and passes it to the method `a.await_suspend(coroutine_handle)`
  - The type of ***a*** must support certain methods
    - Awaitable object or awaiter

co_await example

```
struct Awaiter {
  std::coroutine_handle<> *hp_;
  constexpr bool await_ready() const noexcept { return false; }
  void await_suspend(std::coroutine_handle<> h) { *hp_ = h; }
  constexpr void await_resume() const noexcept {}
};

ReturnObject
counter(std::coroutine_handle<> *continuation_out)
{
  Awaiter a{continuation_out};
  for (unsigned i = 0;; ++i) {
    co_await a;
    // use i here
  }
}
```

```
void main()
{
  std::coroutine_handle<> h;
  counter(&h);
  for() {
    h();
    // unable to get i, just call
  }
  h.destroy();
}
```

What does co_await again (2nd attempt)?

```
auto res = co_await expr;


auto && a = expr;
if(!a.await_ready()) {
  a.await_suspend(coroutine_handle);
  // suspension point
}
auto res = a.await_resume();
```

Predefined awaiters

- Include <coroutine>
  - `std::suspend_always`
    - `await_ready` returns false
  - `std::suspend_never`
    - `await_ready` returns true

Coroutine return object

- Coroutine return type R must be an object with nested type R::promise_type
  - Missing member function causes undefined behavior

```
struct ReturnObject {
  struct promise_type {
    ReturnObject get_return_object() { return {}; }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() { return {}; }
    void unhandled_exception() {}
  };
};
```

What does co_yield?

- We need to get values from coroutines somehow
  - `co_yield e;` is equivalent to `co_await p.yield_value(e);`, where ***p*** is a promise

co_yield example – 1st part

```
struct ReturnObject {
  struct promise_type {
    unsigned value_;

    ReturnObject get_return_object() {
      return { // Uses C++20 designated initializer syntax
        .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
      };
    }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() { return {}; }
    void unhandled_exception() {}
    std::suspend_always yield_value(unsigned value) {
      value_ = value;
      return {};
    }
  };

  std::coroutine_handle<promise_type> h_;
};
```

co_yield example – 2nd part

```
ReturnObject
counter()
{
  for (unsigned i = 0;; ++i)
    co_yield i;       // co yield i => co_await promise.yield_value(i)
}

void
main()
{
  auto h = counter().h_;
  auto &promise = h.promise();
  for (int i = 0; i < 3; ++i) {
    std::cout << "counter: " << promise.value_ << std::endl;
    h();
  }
  h.destroy();
}
```

What does co_return?

- How to signal that the coroutine is complete?
  - Useful for finite streams
  - Coroutine can call `co_return e;` for returning a final value ***e***
    - Compiler inserts `p.return_value(e);`
  - Coroutine can call `co_return;` without value to end the coroutine without a final value
    - Compiler inserts `p.return_void();`
  - Coroutine execution falls off the end of the function
    - Equivalent to the previous case
- Check if coroutine is completed
  - You can call h.done()

co_return example – 1st part

```
struct ReturnObject {
  struct promise_type {
    unsigned value_;

    ~promise_type() { /* do something */ }
    ReturnObject get_return_object() {
      return {
        .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
      };
    }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_always final_suspend() { return {}; }
    void unhandled_exception() {}
    std::suspend_always yield_value(unsigned value) {
      value_ = value;
      return {};
    }
    void return_void() {}
  };

  std::coroutine_handle<promise_type> h_;
};
```

co_return example – 2nd part

```
ReturnObject
counter()
{
  for (unsigned i = 0; i < 3; ++i)
    co_yield i;
  // falling off end of function or co_return;
}

void
main()
{
  auto h = counter().h_;
  auto &promise = h.promise();
  while (!h.done()) { // Do NOT use while(h) (which checks h non-NULL)
    std::cout << "counter: " << promise.value_ << std::endl;
    h();
  }
  h.destroy();
}
```

What about remaining member functions from promise?

- Compiler wraps coroutine function body

```
{
    promise-type promise promise-constructor-arguments ;
    try {
        co_await promise.initial_suspend() ;
        function-body
    } catch ( ... ) {
        if (!initial-await-resume-called)
            throw ;
        promise.unhandled_exception() ;
    }
final-suspend :
    co_await promise.final_suspend() ;
}
```

Automatic clean up

- Trick with p.final_suspend()
  - If `final_suspend` suspends the coroutine, the state remains valid and code outside of the routine is responsible for freeing the object by calling `destroy()`
  - If `final_suspend` does not suspend the coroutine, then the coroutine state will be automatically destroyed