Boost Your Program's Health

by Adding Fibers to your Coroutine

CppCon 2017

David Sackstein

davids@codeprecise.com

Thanks

- Oliver Kowalke:
  - Author of Boost Context, Boost Coroutines and Boost Fiber
  - Author of proposals to ISO/WG21 for stackful coroutines and callcc
- Nat Goodspeed:
  - Contributor to the libraries and proposals to ISO/WG21 for stackful coroutines and callcc
  - Uses the libraries in his work and speaks about these technologies at conferences.

Agenda

- Overview of coroutines
- Two implementations: Stackless and Stackful
- Generators
- Asynchronous APIs
- Threads and Fibers
- The Boost Fiber library
- Questions and Summary

More Information

- Presentation and samples at: https://github.com/david-sackstein/CppCon2017
- Boost Fiber documentation
- My email: davids@codeprecise.com

What are Coroutines?

- Coroutines are a generalization of subroutines.
- Coroutines can be suspended and resumed. The resume at the point of suspension with the same state.

Use Cases for Coroutines

- Generators, Parsers, Pulling from Visitors
- Asynchronous APIs without callbacks
- Combinations of the above: Asynchronous generators
- What is common to all of the above?
  - Coroutines help us “Invert Control”
  - We allow the code that is called in a callback behave as if it were in control

Inversion of Control with Generators

- Both Producer and Consumer want to be in the driver's seat:
  - Producers want to call a function to transfer data
  - Consumers want to call a function to receive data
- With coroutines:
  - The Consumer pulls data. The Producer is the coroutine (inverted):
  - The producer suspends on entry
  - Then pushes data and suspends.
  - Then resumes only when the consumer pulls again.

Two Coroutine Implementations

- Coroutines TS:
  - Stackless
  - Suspend by return
- Boost Coroutine2
  - Stackful
  - Suspend by call

```cpp
#include <boost/coroutine2/all.hpp>

template<typename T>
using pull_type = typename boost::coroutines2::asymmetric_coroutine<T>::pull_type;

template<typename T>
using push_type = typename boost::coroutines2::asymmetric_coroutine<T>::push_type;

static pull_type<int> GenerateIntegers()
{
    pull_type<int> source ([](push_type<int>& sink)
    {
        for (int i = 0; i < 5; i++)
        {
            sink(i);
        }
    });

    return source;
}
```

Generator using Boost Coroutine2

Generator using Coroutine TS

```
#include <experimental/generator>

template <typename T>
using generator = std::experimental::generator<T>;

static generator<int> GenerateIntegers()
{
    for (int i = 0; i < 5; i++)
    {
        co_yield i;
    }
}
```

First class objects?

- The generator is a first class object
  - Therefore generators can be chained together.
  - This is true for both coroutine implementations

Chained Generator using Boost Coroutine2

```
static pull_type<int> GenerateSquares(pull_type<int>& anotherSource)
{
    pull_type<int> source ([&](push_type<int>& sink)
    {
        for (int i : anotherSource)
        {
            sink(i * i);
        }
    });

    return source;
}
```

Chained Generator using Coroutines TS

```
static generator<int> GenerateSquares(generator<int>& anotherSource)
{
    for (int i : anotherSource)
    {
        co_yield i * i;
    }
}
```

First class objects?

- The generator is a first class object
  - Therefore generators can be chained together.
  - This is true for both coroutine implementations
- What about the push capability? Can that be passed around?
  - In Boost Coroutine2 the push_type can be passed around.
  - But in Coroutines TS we do not have access to a push object – the coroutine invokes a key word (co_yield) not a callable object.

Nested Generation with Boost Coroutine2

```
static void ExtractedMethod(
    pull_type<int>& anotherSource,
    push_type<int>& sink)
{
    for (int i : anotherSource)
    {
        sink(i * i);
    }
}

static pull_type<int> GenerateSquares(pull_type<int>& anotherSource)
{
    return pull_type<int> ([&](push_type<int>& sink)
    {
        ExtractedMethod(anotherSource, sink);
    });
}
```

Implications of Stacklessness

- Resumable functions require less memory on initial call and in some cases, the heap allocation can be elided and the code inlined
- However
  - Where generators are nested, nested iterations are required
  - As they suspend by return all functions that invoke a coroutine must be aware that the callee may suspend.

Implications of StackFullness

- Boost Coroutine2 coroutines needs to allocate a stack on the initial call. Still, stack allocation can be customized using the Boost Coroutine2 library.
- Because they suspend by call
  - Suspension is transparent to the caller.
  - This makes it easier to introduce coroutines to legacy code.
  - Nat Goodspeed demonstrates this in his C++Now 2016 talk: Pulling Visitors
  - Another example, a SAX puller

Asynchronous IO

- Not only IO – Any operation that takes time.
- Caller should not block the thread. If it does, an expensive context switch occurs.
- Instead we initiate the IO operation, install a callback and go and do something useful.
- When the operation completes the callback is invoked and we pick up where we were upon suspension.

Problems with callbacks

- Difficult to explain and to maintain
  - The business logic of the class is split up into disjoint functions.
  - State has to be stored and retrieved each time a callback is called.
- Error handling is difficult.
  - In particular exceptions cannot be used
- The problem requires inversion of control.
  - We would like the function and its callback to look like one function and not like a callback

Boost Asio

- Provides a wrapper for synchronous and asynchronous use of sockets (and other OS facilities)
- io_service implements the Proactor Pattern for asynchronous IO.
- Christopher Kohlhoff, the author of the library, provides a generalized way to invoke asynchronous operations.
- Callable objects are only one method of completion. The compiler can deduce different return types from the completion token passed as the last parameter.
- In this way support for other mechanisms are provided such as return a future.

Coroutines and Boost Asio

- The code shows how a boost asio system timer that is invoked asynchronously can be awaited upon.

Coroutine TS implemented coroutine_traits <future<void>, _ArgTypes...>

```
std::future<void> sleepy(io_service &io) {

    io_service::work keep_io_service_alive { io };
    system_timer timer(io);

    timer.expires_from_now(1000ms);

    co_await timer.async_wait(boost::asio::use_future);

    std::cout << "After co_await 1\n";
    timer.expires_from_now(1000ms);

    co_await timer.async_wait(boost::asio::use_future);

    std::cout << "After co_await 2\n";
    timer.expires_from_now(1000ms);

    co_await timer.async_wait(boost::asio::use_future);

    std::cout << "After co_await 3\n";
}

int main() {
    io_service io;
    sleepy(io);
    io.run();
};
```

Keeps io_service busy so it exits only after coroutine ends

Indicates to asio that the operation should return a std::future object.

From Threads to Coroutines To Fibers

- The Echo Server Project
- Four Parts
  - A. AsioSyncThreads
  - B. AsioAsyncCallbacks
  - C. AsioAsyncCoroutines
  - D. AsioAsyncFibers

The Echo Server Project

- Implemented an EchoServer and a few EchoClients
- Clients send a message, Server responds with the same message, Clients send the message they received and so on.
- A Message is defined as a header indicating size and data of the specified size.
- Writing a Message requires to writes and reading a message requires two reads.
- We will zoom in on the Messenger object which reads and writes messages.

A. AsioSyncThreads

- Blocking IO on a thread per connection

Reading a message synchronously

```
Message Messenger::read()
{
    size_t size;

    boost::asio::read(
        _socket,
        boost::asio::buffer(&size, sizeof(size)),
        boost::asio::transfer_all());

    Message message (size);

    boost::asio::read(
        _socket,
        boost::asio::buffer(message.data(), message.size()),
        boost::asio::transfer_all());

    return message;
}
```

A. AsioSyncThreads

- Blocking IO on a thread per connection
- The Good:
  - Main thread listening for connections remains responsive
- The Bad
  - Threads are expensive. Will not scale to thousands.
  - Context switching takes time and will dominate the server for large scales.

B. AsioAsyncCallbacks

- Asynchronous IO using Boost Asio serviced by three threads (arbitrary number). The completion handlers are callback lambdas.

Reading a message asynchronously

```
void Messenger::async_read(std::function<void(const error_code&, Message&&)> handler)
{
    auto callback = [this, handler](const error_code& ec, size_t bytes_transferred)
    {
        if (ec)
            handler(ec, 0);
        else
        {
            _message = Message(_size);

            boost::asio::async_read(
                _socket,
                boost::asio::buffer(_message.data(), _message.size()),
                boost::asio::transfer_all(),
                [this, handler](const error_code ec, size_t bytes_transferred)
                {
                    handler(ec, std::move(_message));
                });
        }
    };

    boost::asio::async_read(
        _socket,
        buffer(&_size, sizeof(_size)),
        transfer_all(),
        callback);
}
```

B. AsioAsyncCallbacks

- Asynchronous IO using Boost Asio serviced by three threads (arbitrary number). The completion handlers are callback lambdas.
- The Good
  - Main thread listening for connections remains responsive
  - Scales well. The number of threads does not depend on the number of connections.
- The Bad
  - Call back code is difficult to debug. Error handling is difficult

C. AsioAsyncCoroutines

- Like B only asynchronous IO has been made to look like synchronous IO due to the use of coroutines.

Asynchronously with Coroutine

```
void EchoConnection::start()
{
    auto self(shared_from_this());

    boost::asio::spawn(_strand, [self, this](yield_context yield)
    {
        self->start(yield);
    });
}
```

Asynchronously with Coroutine

```
Message Messenger::async_read(yield_context yield)
{
    size_t size;

    size_t bytes_transferred =
        boost::asio::async_read(
            _socket,
            buffer(&size, sizeof(size)),
            transfer_all(), yield);

    auto message = Message(size);

    bytes_transferred = boost::asio::async_read(
        _socket,
        boost::asio::buffer(message.data(), message.size()),
        boost::asio::transfer_all(),
        yield);

    return std::move(message);
}
```

C. AsioAsyncCoroutines

- Like B only asynchronous IO has been made to look like synchronous IO due to the use of coroutines.
- The Good
  - Main thread listening for connections remains responsive
  - Scales well. The number of threads does not depend on the number of connections.
  - The code is now organized like synchronous code.

Introducing the Boost Fiber Library

Boost Fiber Library Architecture

- A fiber is a userland thread - a thread of execution that is scheduled cooperatively.
- The library is organized as coroutines plus a manager plus a scheduler.
- Each fiber has its own stack.
- Like Boost Coroutine the library uses Boost Context to allocate and switch between stacks.
- You can choose the stack allocation strategy. The default is fixed size. The size itself is configurable.

Fiber vs Thread

- At most one fiber on a thread can be running
- Therefore a fiber does not need to protect resources from other fibers running on the same thread.
- However, if a fiber on another thread access the resource – protection is required.
- Spawning fibers does not distribute your computation across more hardware cores.
- But fibers do help you manage the of work on one (or more) threads.

Manager

- A fiber can be in the running, suspended or ready state.
- The running fiber can call the manager to yield or suspend itself.
  - When a fiber yields it moves to the ready state
  - When a fiber suspends it moves to a suspended state.
- In both cases the manager uses a scheduling algorithm to select another ready fiber to run.
- The manager performs a context switch to the selected fiber.
- If there were no ready fibers, the manager blocks the thread.
- Context switching is direct (symmetrical), the manager runs on the source fiber.

Scheduler Algorithm

- There is one scheduler algorithm for all fibers in a thread
- The scheduler's responsibility is to pick one of the ready fibers that should to run.
- The default is a round robin scheduler among the ready fibers on the thread.
- So, by default a fiber will always be resumed on the thread where it was created.
- A blocked fiber can however be awoken by another thread

Scheduler Algorithm

- The scheduling algorithm is an extension point. You can also implement and install your own.
- You can also install others provided by the library for instance:
  - shared_work: ready fibers from all threads are treated equally
  - work_stealing: local ready fibers are selected if any, otherwise a fiber is stolen from a scheduler of a different thread
- Note that these algorithms migrate fibers between threads which requires care in protecting shared resources

Fiber Suspension

- A fiber can yield itself or it can suspend itself (a.k.a. block) in a number of ways:
  - It can request to sleep (until a time or for a specified duration)
  - Use fiber synchronization objects that are defined in the library.
- The semantics of the synchronization objects are are similar to those in the std::thread library, however:
- The fiber types suspend only the current fiber. They do not block the thread (unless there are no other fibers to run)

Synchronization Object

Type	Comment
boost::fibers::mutex	Can be used with std::unique_lock
boost::fibers::condition_variable	No spurious wake ups
boost::fibers::barrier	Reusable fiber rendez-vous object
boost::fibers::future<T>, boost::fibers::promise<T> boost::fibers::packaged_task<T>	
boost::fibers::buffered_channel<T> boost::fibers::unbuffered_channel<T>	Like a future-promise of a sequence of values

Context switching is fast*

Haskell | stack-1.4.0/ghc-8.0.1	Go | go1.8.1	Erlang | erts-8.3
0.05 µs - 0.06 µs	0.42 µs - 0.49 µs	0.63 µs - 0.73 µs

Table 1.3. time per thread (average over 10,000 - unable to spawn 1,000,000 threads)

pthread	`std::thread`	`std::async`
54 µs - 73 µs	52 µs - 73 µs	106 µs - 122 µs

Table 1.4. time per fiber (average over 1.000.000)

fiber (16C/32T, work stealing, tcmalloc)	fiber (1C/1T, round robin, tcmalloc)
0.05 µs - 0.09 µs	1.69 µs - 1.79 µs

* Boost Fiber Documentation

Launch a fiber

```
boost::fibers::fiber _fiber(
    boost::fibers::launch::dispatch, // do not enter now
    []()
    {
        std::cout << "Running\n";
    });

boost::this_fiber::yield();
```

Using an unbuffered_channel

```
char WaitForChar()
{
    boost::fibers::unbuffered_channel<char> channel;

    std::thread canceller([&] {
        char c = getchar();
        channel.push(c);
    });

    canceller.detach();

    char c;
    channel.pop(c);
    return c;
}
```

Using a future and a promise

```
boost::fibers::promise<error_code> pr;
boost::fibers::future<error_code> fu = pr.get_future();

boost::asio::async_read(
    _socket,
    buffer(data, size),
    transfer_all(),
    [&](const error_code& ec, size_t bytes_transferred)
    {
        pr.set_value(ec);
    });

return fu.get();
```

D. AsioAsyncFibers

- AsioAsyncFibers
  Like A only fibers are used instead of threads and asynchronous IO is used in combination with boost::fibers::future

Asynchronously with Fiber

```
error_code AsyncMessenger::read(char* data, size_t size)
{
    boost::fibers::promise<error_code> pr;
    boost::fibers::future<error_code> fu = pr.get_future();

    boost::asio::async_read(
        _socket,
        buffer(data, size),
        transfer_all(),
        [&](const error_code& ec, size_t bytes_transferred)
        {
            pr.set_value(ec);
        });

    return fu.get();
}
```

Blocks the current fiber only

D. AsioAsyncFibers

- AsioAsyncFibers
  Like A only fibers are used instead of threads and asynchronous IO is used in combination with boost::fibers::future
- The Good
  - All of the advantages of A-B
- The Bad
  - Both Boost Asio and the Fiber Library have there own “managers”. Coordinating them efficiently is not trival.

Questions

Summary

- We have two implementations of coroutines
  - Stackless (in standardization process) and Stackful (Boost)
- Coroutines enable inversion of control
  - Generators
  - Asynchronous operations
- Threads should be used for parallel computation (or for a small number of background tasks)
- Fibers help organize the execution of asynchronous IO on a thread.