Coroutines, Fibers and Threads, Oh My!

Context for C++ Context Switching

Nat Goodspeed
Linden Lab

Coroutine control transfer

```
// … some logic …
func(…);

// … intermediate processing …

// … resume original logic …
```

```
void func(…)
{
    while (condition)
    {
        …
    }
}
```

Linden Lab®

Symmetric vs. Asymmetric Coroutines

- Symmetric:
  - A → B
  - B → D
  - D → C
  - C → A
- Asymmetric:
  - A → B
  - B → (caller)

Unidirectional data flow

- Caller ← pull_type
- Caller → push_type

A few use cases

- Generators
- Filter chains
- Input/output filtering
- Flattening trees

Boost.Coroutine API

- typedef boost::coroutines::asymmetric_coroutine<T> coro_t;
- Coroutine function accepts coro_t::push_type& sink
- In function body, calls to sink(T) send value to caller
- Caller instantiates coro_t::pull_type, passing coroutine function.
- This runs the function until it calls sink(T) – or returns.
- At that point control returns to caller.
- coro_t::pull_type::operator bool() tests whether there’s a value available.
- coro_t::pull_type::get() retrieves it.
- coro_t::pull_type::operator() passes control back to coroutine function to try for another value.

XML Problem

- Problem: need to recursively parse an XML document.
- Solution: grab any XML DOM parser.
- Problem twist: the document size is completely arbitrary. We cannot assume the DOM will all fit into memory.
- Solution: grab a SAX parser, e.g. "Diego's XML Parser": https://code.google.com/p/die-xml/

Diego’s XML Parser API

- Instantiate xml::sax::Parser
- Specify callbacks of interest
- Call parse() with std::istream
- parse() calls callbacks as interesting events are detected, returns only at end of document
- parse() throws xml::Exception on error

Diego’s XML Parser: startDocument()

- Callback accepts:
  - const xml::sax::Parser::TagType& name
  - xml::sax::AttributeIterator& attrs
- API is designed to avoid unwanted data copying. TagType (aka std::string) is passed by reference.
- Attributes are retrieved via AttributeIterator – that is, only on demand. Even the AttributeIterator is passed by reference.

Diego’s XML Parser: endDocument()

- **Callback accepts:**
  - const xml::sax::Parser::TagType& name

Diego's XML Parser: startTag()

- **Callback accepts:**
  - const xml::sax::Parser::TagType& name
  - xml::sax::AttributeIterator& attrs

Diego's XML Parser: endTag()

- **Callback accepts:**
  - const xml::sax::Parser::TagType& name

Diego's XML Parser: characters()

- Callback accepts:
  - xml::sax::CharIterator& text
- Again, text is only copied on demand: CharIterator can get a char at a time, or a std::string representing the whole text chunk.
- Again, CharIterator is itself passed by reference.

Coroutines

- Oh oh!
- Business logic requires recursive traversal of the XML.
- You can't get there from here! Each time you're done looking at an interesting parsing event, you must return from the callback to the parser. You cannot also build recursive processing. You only have one stack!

Coroutines

- **But what if you had *two?***

Parsing Event Class Hierarchy

```
// Represent a subset of interesting SAX events
struct BaseEvent: public boost::noncopyable
{
    virtual ~BaseEvent() {}
};

// End of document or element
struct CloseEvent: public BaseEvent
{
    // CloseEvent binds (without copying) the TagType reference.
    CloseEvent(const xml::sax::Parser::TagType& name):
        mName(name)
    {}

    const xml::sax::Parser::TagType& mName;
};
```

Parsing Event Class Hierarchy

```
// Start of document or element
struct OpenEvent: public CloseEvent
{
    // In addition to CloseEvent's TagType,
    // OpenEvent binds AttributeIterator.
    OpenEvent(const xml::sax::Parser::TagType& name,
                        xml::sax::AttributeIterator& attrs):
        CloseEvent(name),
        mAttrs(attrs)
    {}

    xml::sax::AttributeIterator& mAttrs;
};
```

Linden Lab

Parsing Event Class Hierarchy

```
// text within an element
struct TextEvent: public BaseEvent
{
    // TextEvent binds the CharIterator.
    TextEvent(xml::sax::CharIterator& text):
        mText(text)
    {}

    xml::sax::CharIterator& mText;
};
```

Linden Lab®

Coroutine Definition

```
// The parsing coroutine instantiates BaseEvent subclass
// instances and successively shows them to the main program.
// It passes a reference so we don't slice the subclass.
typedef boost::coroutines::asymmetric_coroutine
                                       <const BaseEvent&> coro_t;
```

Coroutine Body

```
void parser(coro_t::push_type& sink, std::istream& in)
{
    xml::sax::Parser xparser;

    // startDocument() will send OpenEvent
    xparser.startDocument(
        [&sink](const xml::sax::Parser::TagType& name,
                xml::sax::AttributeIterator& attrs)
        {
            sink(OpenEvent(name, attrs));
        });
```

Linden Lab®

Coroutine Body

```
void parser(coro_t::push_type& sink, std::istream& in)
{
    xml::sax::Parser xparser;
    …
    // startTag() will likewise send OpenEvent
    xparser.startTag(
        [&sink](const xml::sax::Parser::TagType& name,
                xml::sax::AttributeIterator& attrs)
        {
            sink(OpenEvent(name, attrs));
        });
```

Coroutine Body

```
void parser(coro_t::push_type& sink, std::istream& in)
{
    xml::sax::Parser xparser;
    …
    // endTag() will send CloseEvent
    xparser.endTag(
          [&sink](const xml::sax::Parser::TagType& name)
          {
             sink(CloseEvent(name));
          });
    // endDocument() will likewise send CloseEvent
    xparser.endDocument(
          [&sink](const xml::sax::Parser::TagType& name)
          {
             sink(CloseEvent(name));
          });
```

Coroutine Body

```
void parser(coro_t::push_type& sink, std::istream& in)
{
    xml::sax::Parser xparser;
    …
    try
    {
        // parse the document, firing registered callbacks
        xparser.parse(in);
    }
    catch (xml::Exception e)
    {
        // xml::sax::Parser throws xml::Exception. Helpfully translate
        // the name and provide it as the what() string.
        throw std::runtime_error(exception_name(e));
    }
}
```

Linden Lab®

Processing Code

```
// Recursively traverse the incoming XML document on the fly,
// pulling BaseEvent& references from 'events'.
// 'indent' illustrates the level of recursion.
// Each time we're called, we've just retrieved an OpenEvent from
// 'events’; accept that OpenEvent as a param.
// Return the CloseEvent that ends this element.
const CloseEvent&
process(coro_t::pull_type& events, const OpenEvent& context,
            const std::string& indent="")
{
  // Capture tag name: as soon as we advance the parser, the
  // TagType& reference bound in OpenEvent will be invalidated.
  xml::sax::Parser::TagType tagName = context.mName;
```

Processing Code

```
const CloseEvent&
process(coro_t::pull_type& events, const OpenEvent& context,
            const std::string& indent="")
{
   …
   // Since the OpenEvent is still the current value from 'events',
   // pass control back to 'events' until the next event. Of course,
   // each time we come back we must check for the end of the
   // results stream.
   while (events())
   {
      // Another event is pending; retrieve it.
      const BaseEvent& event = events.get();
```

Linden Lab

Processing Code

```
const CloseEvent&
process(coro_t::pull_type& events, const OpenEvent& context,
              const std::string& indent="")
{
    …
    while (events())
    {
        const BaseEvent& event = events.get();
        …
        const OpenEvent* oe;
        if ((oe = dynamic_cast<const OpenEvent*>(&event)))
        {
            // When we see OpenEvent, recursively process it.
            process(events, *oe, indent + "    ");
        }
```

Processing Code

```
while (events())
{
    const BaseEvent& event = events.get();
    const CloseEvent* ce;
    …
    else if ((ce = dynamic_cast<const CloseEvent*>(&event)))
    {
        // When we see CloseEvent, validate its tag name and
        // then return it. (This assert is really a check on
        // xml::sax::Parser, since it already validates matching
        // open/close tags.)
        assert(ce->mName == tagName);
        return *ce;
    }
```

Processing Code

```
while (events())
{
    const BaseEvent& event = events.get();
    const TextEvent* te;
    …
    else if ((te = dynamic_cast<const TextEvent*>(&event)))
    {
        // When we see TextEvent, just report its text, along with
        // indentation indicating recursion level.
        std::cout << indent << "text: '" << te->mText.getText()
                   << "'\n";
    }
}
```

Linden Lab®

Processing Code

```
void parse(const std::string& doc)
{
    std::cout << "\nParsing:\n" << doc << '\n';
    std::istringstream in(doc);


        coro_t::pull_type events(boost::bind(parser, _1, boost::ref(in)));









}
```

Coroutine Body

```
void parser(coro_t::push_type& sink, std::istream& in)
{
    xml::sax::Parser xparser;
    …
    try
    {
        // parse the document, firing registered callbacks
        xparser.parse(in);
    }
    catch (xml::Exception e)
    {
        // xml::sax::Parser throws xml::Exception. Helpfully translate
        // the name and provide it as the what() string.
        throw std::runtime_error(exception_name(e));
    }
}
```

Processing Code

```
void parse(const std::string& doc)
{
    std::cout << "\nParsing:\n" << doc << '\n';
    std::istringstream in(doc);


        coro_t::pull_type events(boost::bind(parser, _1, boost::ref(in)));
        assert(events);
        // This dynamic_cast<&> is itself an assertion that the first event is an
        // OpenEvent.
        const OpenEvent& context =
            dynamic_cast<const OpenEvent&>(events.get());
        process(events, context);




}
```

Linden Lab

Processing Code

```
// Recursively traverse the incoming XML document on the fly,
// pulling BaseEvent& references from 'events'.
// 'indent' illustrates the level of recursion.
// Each time we're called, we've just retrieved an OpenEvent from
// 'events'; accept that OpenEvent as a param.
// Return the CloseEvent that ends this element.
const CloseEvent&
process(coro_t::pull_type& events, const OpenEvent& context,
            const std::string& indent="")
{
    …
}
```

Linden Lab

Processing Code

```
void parse(const std::string& doc)
{
    std::cout << "\nParsing:\n" << doc << '\n';
    std::istringstream in(doc);
    try
    {
        coro_t::pull_type events(boost::bind(parser, _1, boost::ref(in)));
        assert(events);
        // This dynamic_cast<&> is itself an assertion that the first event is an
        // OpenEvent.
        const OpenEvent& context =
            dynamic_cast<const OpenEvent&>(events.get());
        process(events, context);
    }
    catch (std::exception& e)
    {
        std::cout << "Parsing error: " << e.what() << '\n';
    }
}
```

Output

```
Parsing:
<root attr="17">
    <text style="important">
        The textile industry is <i>extremely <b>important</b></i>.
    </text>
</root>

    text: 'The textile industry is '
        text: 'extremely '
            text: 'important'
    text: '.
    '
```

Output

Parsing:
<root attr="17">
<text style="important">
The textile industry is <i>extremely important</i>.
</test>
</root>

text: 'The textile industry is '
text: 'extremely '
text: 'important'
text: '.
'

Parsing error: TAG_MISMATCH

Why a Coroutine?

- “You could do this with threads.”
- True enough…
- But do you really want to?
- With a SAX parser, control passes back and forth between the parser and the business logic. It makes no sense for either to advance ahead of the other.

Using threads…

- You would have to pass BaseEvent subclass objects from the parser thread to the consumer thread using a thread-safe queue.
- The parser instantiates one of several BaseEvent subclass objects. Because it's polymorphic, to pass it on a queue you would have to allocate it on the heap and store pointers in the queue.
- BaseEvent subclasses bind references to transient parser objects. You would have to make (e.g.) CloseEvent store a copy of its TagType instead of binding a reference.
- But for attributes and text, xml::sax::Parser passes an AttributeIterator or CharIterator. It's not enough to copy the iterator: you must copy all of the referenced data.
- That would be all the data in the XML document.

Using coroutines…

- The parser coroutine instantiates a temporary stack BaseEvent subclass object.
- It passes the business logic a const reference to *that temporary.*
- The temporary binds references to transient SAX parser objects.
- An AttributeIterator or CharIterator is dereferenced only when needed. All this takes place during the SAX callback.
- No superfluous data copying!

But wait, there's more!

- What about SAX parser exceptions?
- With threads, you would have to:
  - catch the exception
  - convert to exception_ptr
  - arrange a way to pass exception_ptr via your thread-safe queue
  - notice the exception_ptr and rethrow the exception.
- With coroutines, you:
  - catch the exception.

Questions so far?

Coroutines Invert Control Flow

- Coroutines can be used to transform a push callback into a pull request.
- What about async I/O?

Network I/O Problem

- Using the same tactic as for the SAX parser, let's write a simple coroutine to "pull" async completion handler results.

Coroutine Body

```
typedef boost::coroutines::asymmetric_coroutine<std::string>
    coro_t;

void pull_data(coro_t::push_type& sink,
                       boost::asio::ip::tcp::socket& socket)
{
    char buffer[1024];
    socket.async_read_some(
        boost::asio::buffer(buffer),
        [&sink](const boost::system::error_code& ec, size_t length)
        {
            sink(std::string(buffer, length));
        });
}
```

Oh oh

```
typedef boost::coroutines::asymmetric_coroutine<std::string>
    coro_t;

void pull_data(coro_t::push_type& sink,
                       boost::asio::ip::tcp::socket& socket)
{
    char buffer[1024];
    socket.async_read_some(
        boost::asio::buffer(buffer),
        [&sink](const boost::system::error_code& ec, size_t length)
        {
            sink(std::string(buffer, length));
        });
}
```

- **Does anybody see a problem here??**

Compare:

```
void parser(coro_t::push_type& sink, std::istream& in)
{
    xml::sax::Parser xparser;
    // … register callbacks …
    // parse the document, firing registered callbacks
    xparser.parse(in);
}
```

- **In this case, the xparser.parse() call *does not return* until it has finished processing the document.**
- **It is completely legitimate for the callbacks to bind a reference to sink because, once xparser.parse() returns, no further callbacks will be invoked.**

Compare:

```
void pull_data(coro_t::push_type& sink,
                      boost::asio::ip::tcp::socket& socket)
{
    char buffer[1024];
    socket.async_read_some(
        boost::asio::buffer(buffer),
        [&sink](const boost::system::error_code& ec, size_t length)
        {
            sink(std::string(buffer, length));
        });
    // block somehow??
}
```

What do we mean by “block”?

- We could call boost::asio::basic_stream_socket::read_some() and block the whole thread. But we specifically want async I/O.
- In fact, we want a fire-and-forget coroutine. We want control to return to the caller while the I/O is pending, but we don’t want the coroutine to exit.
- We want someone else to manage the coroutine’s lifespan. Once we’ve launched it, we want to be able to return through arbitrary levels of call depth without destroying the coroutine instance.
- To service the pending async I/O request, we must arrange to pass control to the related io_service instance while waiting.

Scaling up

- An asymmetric coroutine has a very specific relationship with the code that launched it.
- What if coroutine A needs to launch coroutine B? To whom does control return?
- What if coroutine A isn't ready to resume when coroutine B blocks?
- What we need is a scheduler.

Boost.Asio coroutine support

- You're just about to point out that Boost.Asio *does* support Boost.Coroutine. How?
- boost::asio::io_service *is* a scheduler for pending async I/O operations.
- The coroutine is owned by its own completion handler.
- A completion handler is owned by the corresponding pending async operation.
- The pending operations are owned by the io_service.

For everything else, there's…

- Boost.Fiber builds on Boost.Coroutine by adding a scheduler + synchronization operations.
- Fiber emulates the std::thread API, but with user-space context switching based on Coroutine.
- There's a separate instance of the Fiber scheduler for each thread on which you launch one or more fibers.
- When you detach() a fiber, the scheduler owns the fiber instance.
- When a fiber blocks, it passes control to the scheduler to dispatch another ready-to-run fiber.
- (Yes, you can replace the default scheduler with logic of your own.)

Fiber Body – simulate blocking I/O

```
std::string pull_data(boost::asio::ip::tcp::socket& socket)
{
    boost::fibers::promise<std::string> promise;
    boost::fibers::future<std::string> future(promise.get_future());
    char buffer[1024];

    socket.async_read_some(
        boost::asio::buffer(buffer),
        [&promise, &buffer]
        (const boost::system::error_code& ec, size_t length)
        {
            promise.set_value(std::string(buffer, length));
            // or promise.set_exception() as appropriate
        });
    return future.get();  // <= blocks here
}
```

Linden Lab®

Fiber Body – Boost.Asio support

```
std::string pull_data(boost::asio::ip::tcp::socket& socket)
{



    char buffer[1024];
    std::size_t length =
    socket.async_read_some(
        boost::asio::buffer(buffer),
        boost::fibers::asio::yield); // <= blocks here






    return std::string(buffer, length);
}
```

Linden Lab

New async completion behavior for Asio

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3964.pdf describes how to present an async operation function such that the *caller* can specify whether to block, return a future, call a callback, etc. The author, Christopher Kohlhoff, proposes this convention for any library capable of initiating an async operation.

Encapsulating the Gory Details

```
std::string pull_data(boost::asio::ip::tcp::socket& socket)
{
    boost::fibers::promise<std::string> promise;
    boost::fibers::future<std::string> future(promise.get_future());
    char buffer[1024];

    socket.async_read_some(
        boost::asio::buffer(buffer),
        [&promise, &buffer]
        (const boost::system::error_code& ec, size_t length)
        {
            promise.set_value(std::string(buffer, length));
            // or promise.set_exception() as appropriate
        });
    return future.get();  // <= blocks here
}
```

Encapsulating the Gory Details

```
void business_logic(boost::asio::ip::tcp::socket& socket)
{
    // …send request on socket…
    std::string response = pull_data(socket);
    // …send next request…
    response = pull_data(socket);
    // …
}
```

Why not separate threads?

- UI thread
- WinRT asynchronous operations
- Legacy code touching static data
- Performance/scaling…

Questions so far?

Linden Lab®

then() what?

- **N3721 (folded into D3904) proposes future::then() as a mechanism for chaining async operations: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3721.pdf**

then() what?

```
using boost::asio::buffer;
char buff[1024];
std::async(sock0.read_some(buffer(buff)))
    .then([](std::future<std::size_t> fsize)
            {
                sock1.write_some(buffer(buff, fsize.get()));
            })
    .then([](std::future<std::size_t>)
            {
                sock0.read_some(buffer(buff));
            })
    .then([](std::future<std::size_t> fsize)
            {
                sock1.write_some(buffer(buff, fsize.get()));
            });
```

Linden Lab

then() what?

- Introduces a DSL for sequencing async operations
  - What happens when you need conditionals?
  - Looping?

One blogger's opinion

- http://paoloseverini.wordpress.com/2014/04/22/async-await-in-c/
- "In particular, .NET's TPL has the defect of making the code overly complex.
- "It is difficult to write *robust*, *readable* and *debuggable* code with TPL. It is too easy to end up with spaghetti code: a large number of Task<T>s interconnected in inextricable chains of continuations.
- "I found out the hard way, when I was given, by my last employer, a .NET component to maintain which had been architected as an incredibly convoluted graph of interconnected TPL Tasks, often joined with conditional *ContinueWith()*.

One blogger's opinion

- http://paoloseverini.wordpress.com/2014/04/22/async-await-in-c/
- "It is particularly difficult to write iterations and branches with tasks.
- "It turns out that writing a 'task loop' correctly is not trivial, especially if there are many possible error cases or exceptions to handle. It is actually so tricky that there are MSDN pages that explains what to do, and what the possible pitfalls are…"

If not then()…

- C++ already has familiar, readable ways to express control flow – hey, let’s use those!

This code isn't robust!

```
using boost::asio::buffer;
char buff[1024];
std::async(sock0.read_some(buffer(buff)))
    .then([](std::future<std::size_t> fsize)
            {
                sock1.write_some(buffer(buff, fsize.get()));
            })
    .then([](std::future<std::size_t>)
            {
                sock0.read_some(buffer(buff));
            })
    .then([](std::future<std::size_t> fsize)
            {
                sock1.write_some(buffer(buff, fsize.get()));
            });
```

Linden Lab

Something of an improvement

```
using boost::asio::buffer;
char buff[1024];

for (;;) {
    std::size_t bytes_read = sock0.async_read_some(
        buffer(buff),
        boost::fibers::asio::yield);
    char* read_end = buff + bytes_read;
    char* write_ptr = buff;
    while (write_ptr < read_end) {
        write_ptr += sock1.async_write_some(
            buffer(write_ptr, (read_end - write_ptr)),
            boost::fibers::asio::yield);
    }
}
```

Cross-Fiber Communication

- The proposed Boost.Fiber library includes synchronization primitives based on std::thread:
  - mutex
  - barrier
  - condition_variable
  - future, promise, packaged_task
  - bounded_queue, unbounded_queue

Cross-Fiber Communication

- As a simple example, consider the unbounded fiber-safe queue: a queue which, when empty, blocks the consuming fiber, permitting other fibers on the same thread to continue executing.
- We can launch multiple fibers to produce results into such a queue…
- We can model “when any” by waiting only for the first such result, ignoring all others.
- We can model “when all” by consuming the result from each producer.

N3785 Executors

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3785.pdf

Asio-style Executors

- https://github.com/chriskohlhoff/executors

Questions?