Document No: N4287
Date: 2014-11-18

Threads, Fibers and Coroutines

slide deck from the Coroutine evening session in Urbana 2014

Gor Nishanov (gorn@nicrosoft.com)

Threads, Fibers and Coroutines

- Thread
  - State: User-mode stack + kernel mode stack + context
  - Run by an OS scheduler
  - Unit of suspension: entire thread, CPU is free to run something else
  - Context: ~ entire register file +
- Fiber (aka User-Mode-Scheduled-Thread, stackful coro) N3985
  - State: User-mode stack + Context
  - Run by some thread
  - Unit of suspension: fiber, underlying thread is free to run
  - Context: ABI mandated non-volatile regs +
- Coroutine (Stackless) N4134, N4244
  - State: Local variables + Context
  - Run by some thread or fiber
  - Unit of suspension: coroutine, underlying thread/fiber is free to run
  - Context: ~ 4 bytes +

N4134: RESUMABLE FUNCTIONS V2

REACTIVE STREAMS MEET COROUTINES

Source

Produces 0.1.2.3...
each 1ms

```
async_stream<int> Ticks()
{
    for (int tick = 0;; ++tick)
    {
        yield tick;
        await sleep_for(1ms);
    }
}
```

SNEAK PEEK (more later)

Transformer

Transforms stream of $v_1.v_2.v_3...$
into a stream of
$(v_1,t_1).(v_2,t_2).(v_3,t_3)...$
where t_i is a timestamp of when
v_i was received

```
template<class T>
auto AddTimestamp(AsyncStream<T> & S)
{
    for await(v: S) yield make_pair(v,system_clock::now());
}
```

Sink

Reduces an asynchronous
stream to a sum of its values

```
future<int> Sum(AsyncStream<int> & input)
{
    int sum = 0;
    for await(v: input)
        sum += v;
    return sum;
}
```

N4134: RESUMABLE FUNCTIONS V2

COMMON PATTERN FOR ASYNC AND SYNC I/O

SNEAK PEEK (more later)

async

```
future<int> tcp_reader(int total)
{
    char buf[64 * 1024];
    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    for (;;)
    {
        auto bytesRead = await conn.read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

sync

```
expected<int> tcp_reader(int total)
{
    char buf[64 * 1024];
    auto conn = await Tcp::Connect("127.0.0.1", 1337, block);
    for (;;)
    {
        auto bytesRead = await conn.read(buf, sizeof(buf), block);
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

N4134: RESUMABLE FUNCTIONS V2

GENERATORS AND ITERABLES AND AGGREGATE INITIALIZATION

generators

```
generator<char> hello() {
   for (ch: "Hello, world\n") yield ch;
}
int main() {
   for (ch : hello()) cout << ch;
}
```

SNEAK PEEK (more later)

Not in N4134

constexpr generators

```
int a[] = { []{ for(int x = 0; x < 10; ++x) yield x*x; } };
```

Equivalent to

```
int a[] = { 0,1,4,9,16,25,36,49,64,81 };
```

Recursive Generators

Checks if in-order depth first traversal of two trees yields the same sequence of values

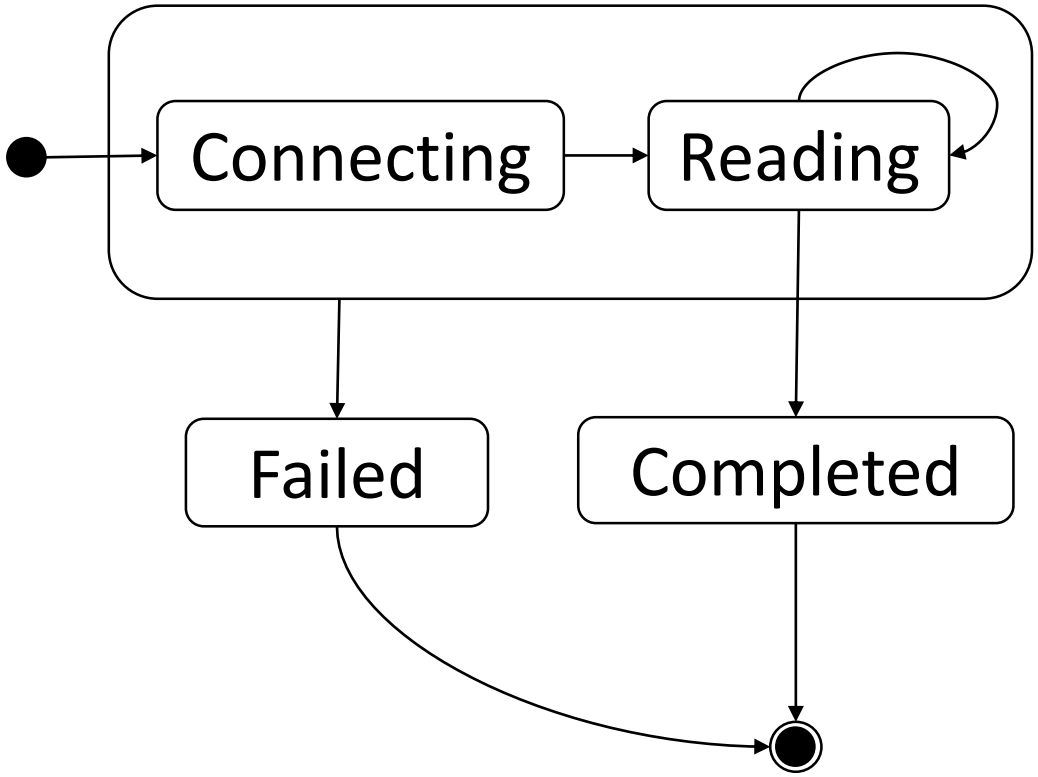
```
auto flatten(node* n) -> generator<decltype(n->value)> {
    if (n == nullptr) return;
    yield flatten(n->left);
    yield n->value;
    yield flatten(n->right);
}
bool same_fringe(node* tree1, node* tree2)
{
    auto seq1 = flatten(tree1);
    auto seq2 = flatten(tree2);
    return equal(begin(seq1), end(seq1),
                 begin(seq2), end(seq2));
}
```

Q: How to come up with generic zero-overhead abstractions?

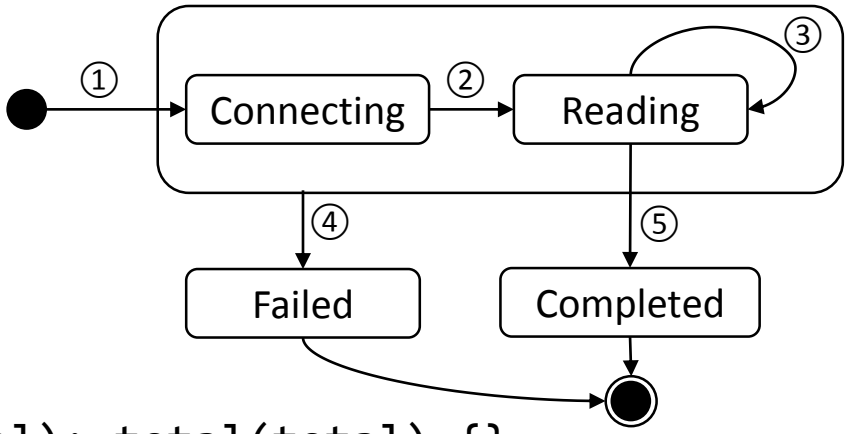
Alex Stepanov:

1. Start with the best known solution solving an important problem on a particular hardware.
2. Think of an abstraction that can capture the pattern of that solution and make it safe and repeatable
3. Recode the original problem, check that no overhead was introduced
4. See if you can lessen the requirements and make it more generic
5. Test applicability to other problems (go to step 1)

Async state machine



Hand-crafted async state machine (1/3)




```
class tcp_reader
{
    char buf[64 * 1024];
    Tcp::Connection conn;
    promise<int> done;
    int total;

    explicit tcp_reader(int total): total(total) {}

  ② void OnConnect(error_code ec, Tcp::Connection newCon);
  ③ void OnRead(error_code ec, int bytesRead);
  ④ void OnError(error_code ec);
  ⑤ void OnComplete();

public:
  ① static future<int> start(int total);
};

int main() {
   cout << tcp_reader::start(1000 * 1000 * 1000).get(); }
```

Hand-crafted async state machine (2/3)

```
future<int> tcp_reader::start(int total) {
   auto p = make_unique<tcp_reader>(total);
   auto result = p->done.get_future();
   Tcp::Connect("127.0.0.1", 1337,
      [raw = p.get()](auto ec, auto newConn) {
            raw->OnConnect(ec, std::move(newConn));
      });
   p.release();
   return result;
}

void tcp_reader::OnConnect(error_code ec,
                           Tcp::Connection newCon)
{
   if (ec) return OnError(ec);
   conn = std::move(newCon);
   conn.Read(buf, sizeof(buf),
      [this](error_code ec, int bytesRead)
         { OnRead(ec, bytesRead); });
}
```

Hand-crafted async state machine (3/3)

```
void tcp_reader::OnRead(error_code ec, int bytesRead) {
   if (ec) return OnError(ec);
   total -= bytesRead;
   if (total <= 0 || bytesRead == 0) return OnComplete();
   conn.Read(buf, sizeof(buf),
      [this](error_code ec, int bytesRead) {
         OnRead(ec, bytesRead); });
}

void OnError(error_code ec) {
   auto p = unique_ptr<tcp_reader>(this);
   done.set_exception(make_exception_ptr(system_error(ec)));
}

void OnComplete() {
   auto p = unique_ptr<tcp_reader>(this);
   done.set_value(total);
}
```

Rewritten as N4134 Coroutine

```
future<int> tcp_reader(int total)
{
    char buf[64 * 1024];
    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    for (;;)
    {
        auto bytesRead = await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}

int main() { cout << tcp_reader(1000 * 1000 * 1000).get(); }
```

Reminder what it looked before

```
class tcp_reader
{
    char buf[64 * 1024];
    Tcp::Connection conn;
    promise<void> done;
    int total;

    explicit tcp_reader(int total): total(total) {}

    void OnConnect(error_code ec, Tcp::Connection newCon);
    void OnRead(error_code ec, int bytesRead);
    void OnError(error_code ec);
    void OnComplete();

public:
    static future<void> start(int total);
};

int main() {
   cout << tcp_reader::start(1000 * 1000 * 1000).get(); }
```

Yeah, pretty, but what about perf?

Yeah, pretty, but what about perf?

	Hand-crafted	N4134
Mbps (5 runs average)	21466.77	21477.13
Binary size (bytes)	362,496	360,448 -2048
allocations	**15,260**	1

Negative-overhead abstraction!

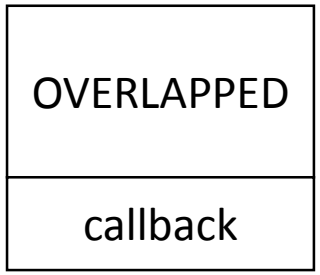
	Hand-crafted	N4134
Mbps (5 runs average)	21466.77	21477.13
Binary size (bytes)	362,496	360,448 -2048
allocations	**15,260**	1

15,260 allocations, How? Why?

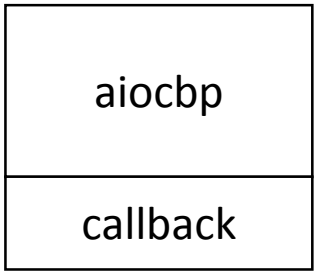
15,260 = 1 + 1 + 15258 = 1 + 1 + 1,000,000,000 / 64K

```
conn.Read(buf, sizeof(buf),
      [this](error_code ec, int bytesRead)
         { OnRead(ec, bytesRead); });

template <class Cb>
void Read(void* buf, size_t bytes, Cb && cb);
```

Windows: ReadFile(fd, ..., OVERLAPPED*)



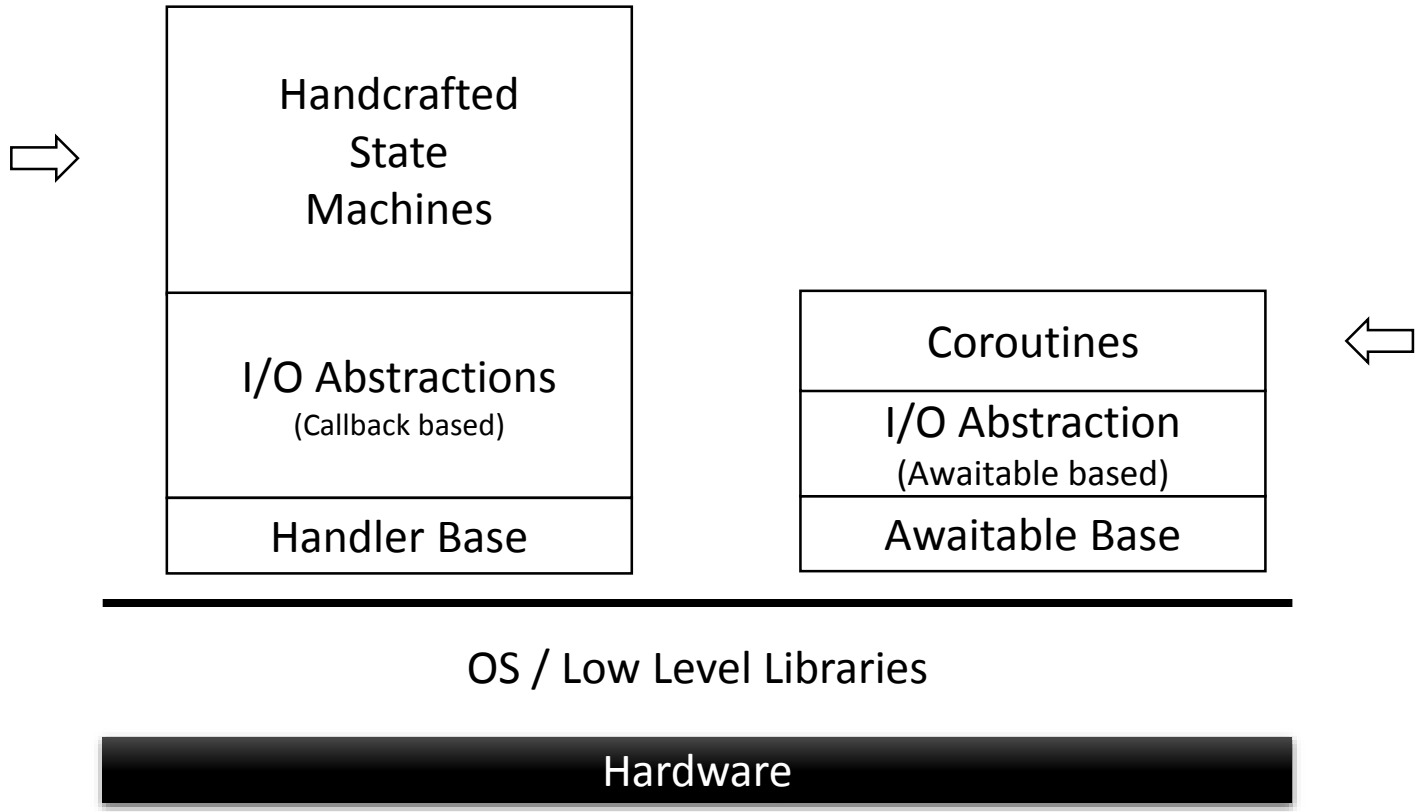
Posix aio: aio_read(fd, aiocbp*)



- Callback pattern leads to code bloat
- Retains inherent inefficiency of allocation of a context for every async op

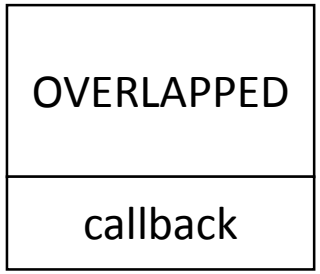
Note: Same problem with boost::asio, N4243 Networking Proposal, N4045 Foundation for async, N4046, N4143: Executors

Coroutines are closer to the metal



Callback machinery (1/3)
(common for all I/O operations)

```
struct OverlappedBase: OVERLAPPED  {
   virtual void Invoke() (ULONG ec, ULONG_PTR nBytes) = 0;
   virtual ~OverlappedBase() {}
};

static void __stdcall io_complete_callback(
   PTP_CALLBACK_INSTANCE, PVOID,
   PVOID Overlapped,
   ULONG IoResult,
   ULONG_PTR NumberOfBytesTransferred,
   PTP_IO)
{
   auto o = reinterpret_cast<OVERLAPPED*>(Overlapped);
   auto me = static_cast<OverlappedBase*>(o);
   me->Invoke(IoResult, NumberOfBytesTransferred);
}
```



Callback machinery (2/3)

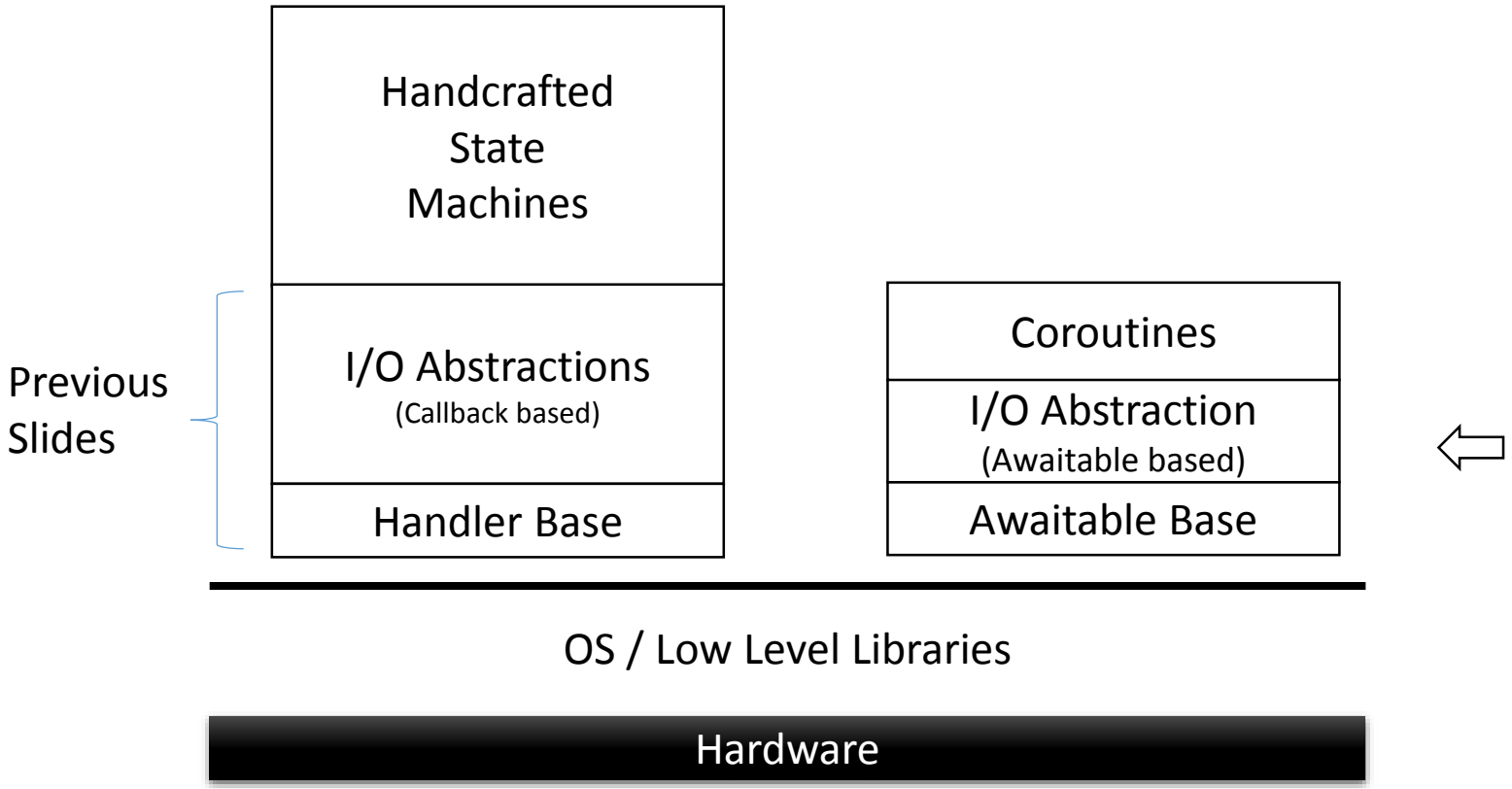
```
template <typename Fn>
struct CompletionWithSizeT : OverlappedBase, private Fn
{
   CompletionWithSizeT(Fn fn): Fn(move(fn)){}

   void Invoke(ULONG ec, ULONG_PTR count) override
   {
      Fn::operator()(
         error_code(ec, system_category()), count);
   }
};

template <typename Fn>
unique_ptr<OverlappedBase> make_handler_with_size_t(Fn && fn)
{
   return make_unique<CompletionWithSizeT<
      decay_t<Fn>>>(forward<Fn>(fn));
}
```

Callback machinery (3/3)

```
template <typename Cb>
void Read(void* buf, size_t bytes, Cb && cb) {
   Read(buf, bytes,
                make_handler_with_size_t(std::forward<Cb>(cb)));
}

void Read(void* buf, size_t size, unique_ptr<OverlappedBase> o)
{
    StartThreadpoolIo(io);
    auto error = TcpSocket::Read(handle, buf, size, o.get());
    if (error) {
       CancelThreadpoolIo(io);
       o->operator()(error, 0);
    }
    o.release();
}
```

```
// sometime during connection construction
io = CreateThreadpoolIo(handle, &io_complete_callback, nullptr, nullptr);
```

Coroutines are closer to the metal



REMEMBER THIS?

Callback machinery (1/3)

(common for all I/O operations)

```
struct OverlappedBase: OVERLAPPED  {
   virtual void Invoke() (ULONG ec, ULONG_PTR nBytes) = 0;
   virtual ~OverlappedBase() {}


   static void __stdcall io_complete_callback(
      PTP_CALLBACK_INSTANCE, PVOID,
      PVOID Overlapped,
      ULONG IoResult,
      ULONG_PTR NumberOfBytesTransferred,
      PTP_IO)
   {
      auto o = reinterpret_cast<OVERLAPPED*>(Overlapped);
      auto me = static_cast<OverlappedBase*>(o);



      me->Invoke(IoResult, NumberOfBytesTransferred);
   }
};
```

Awaitable: Overlapped Helper (1/2)

```
struct OverlappedBase: OVERLAPPED  {
   coroutine_handle<> Invoke;
   ULONG_PTR nBytes;
   ULONG ec;

   static void __stdcall io_complete_callback(
      PTP_CALLBACK_INSTANCE, PVOID,
      PVOID Overlapped,
      ULONG IoResult,
      ULONG_PTR NumberOfBytesTransferred,
      PTP_IO)
    {
      auto o = reinterpret_cast<OVERLAPPED*>(Overlapped);
      auto me = static_cast<OverlappedBase*>(o);

      me->ec = IoResult;
      me->nBytes = NumberOfBytesTransferred;
      me->Invoke();
    }
};
```

```
mov rcx, [rcx]
call [rcx]
```

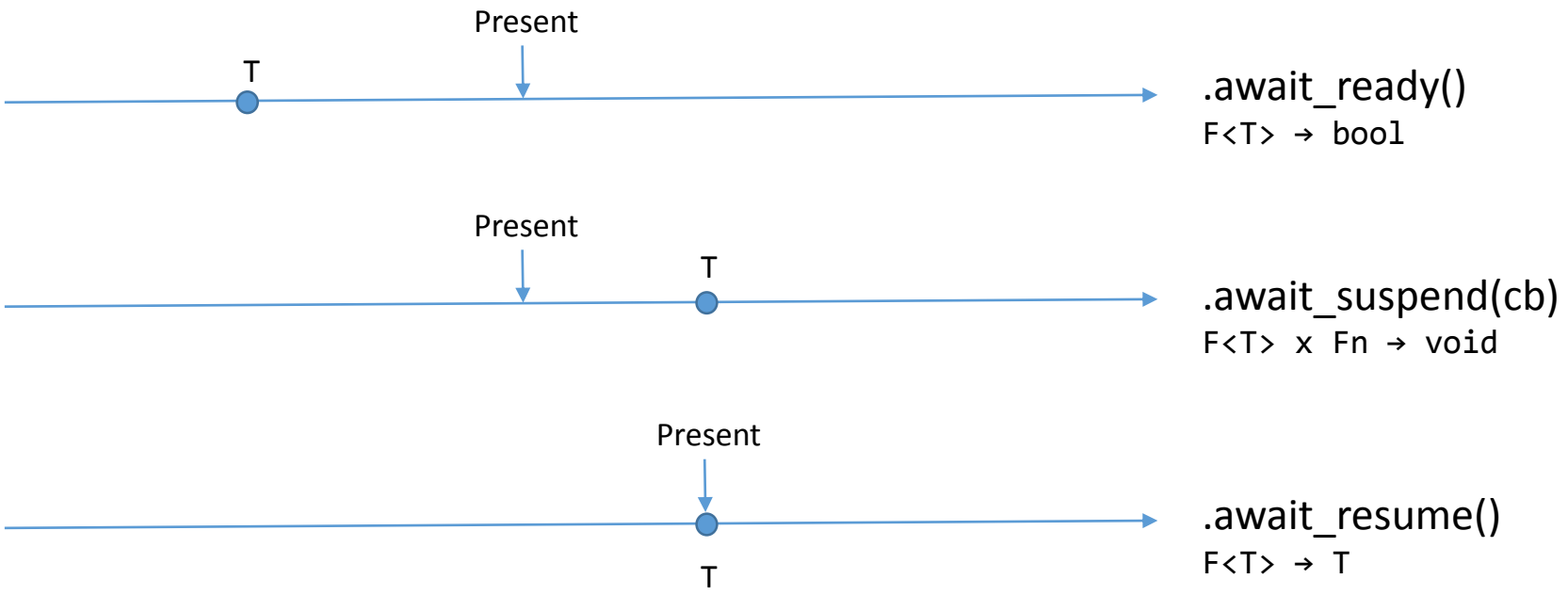
What are we awaiting upon?

```
future<int> tcp_reader(int total)
{
    char buf[64 * 1024];
    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    for (;;)
    {
        auto bytesRead = await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

2 x 2 x 2

- Two new keywords
  - **await**
  - **yield**
    syntactic sugar for: **await** $p.yield_value(expr)
- Two new concepts
  - Awaitable
  - Coroutine Promise
- Two library types
  - coroutine_handle
  - coroutine_traits

Awaitable – Concept of the Future<T>

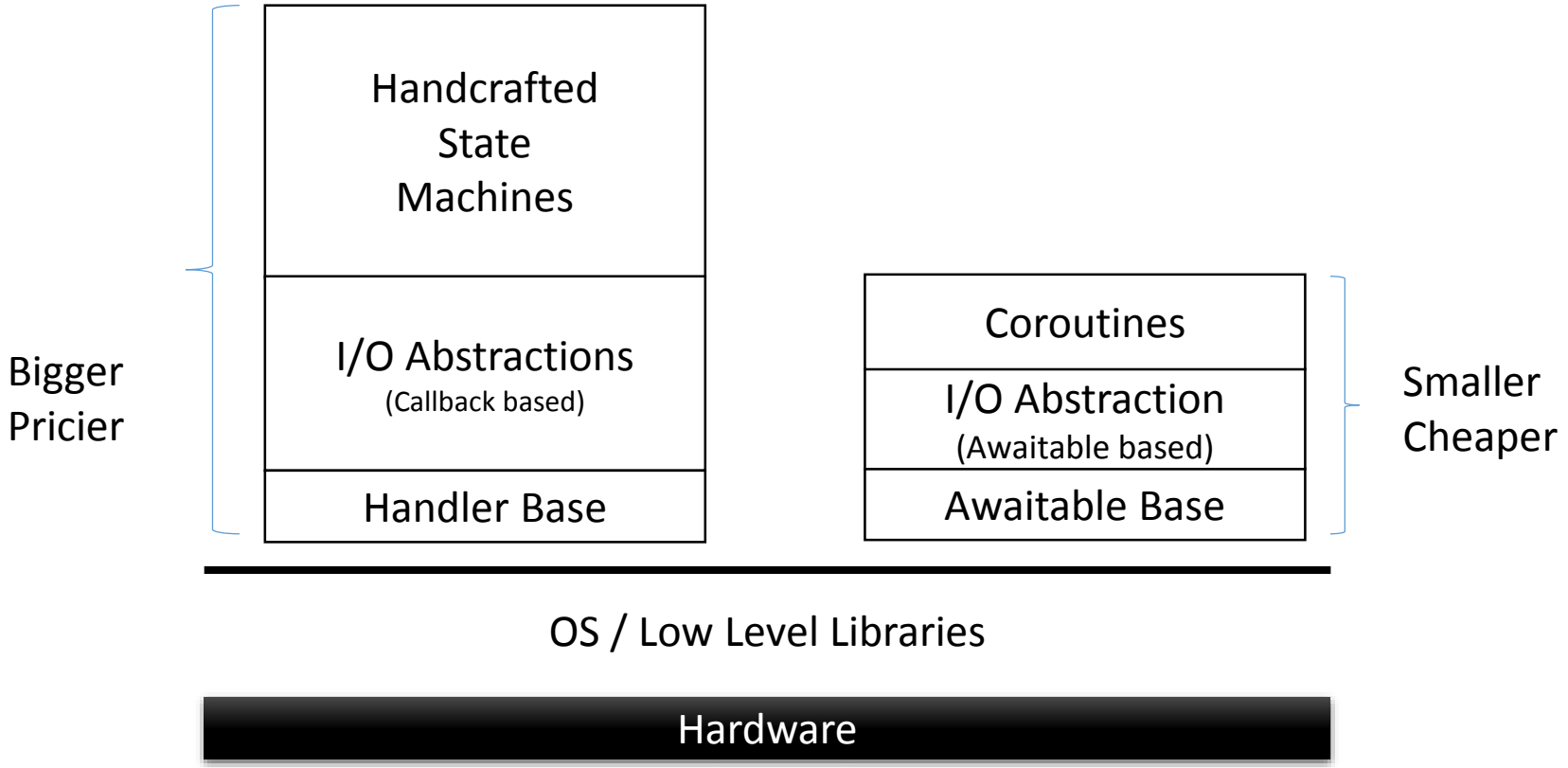



```
await expr-of-awaitable-type
```

Awaitable: Read (2/2)

```
auto Connection::Read(void* buf, size_t bytes) {
   struct awaiter: OverlappedBase {
      void* buf;
      size_t size;
      Connection * my;
      bool await_ready() const { return false; }
      void await_suspend(coroutine_handle<> cb) {
         Invoke = cb;
         StartThreadpoolIo(my->io);
         auto err = TcpSocket::Read(my->handle, buf, size, this);
         if (err) {
            CancelThreadpoolIo(my->io); throw system_error(err);}
         }
      }
      int await_resume() {
         if (ec) throw system_error(ec);
         return nBytes;
      }
    };
    return awaiter{ buf, bytes, this };
}
```

Coroutines are closer to the metal



`await` <expr>

Expands into an expression equivalent of

```
{
  auto && tmp = <expr>;
  if (!await_ready(tmp)) {
     await_suspend(tmp, <resume-function-object>);
  }
  return await_resume(tmp);
}
```

`await` <expr>

If await_suspend return type is not void, then

```
{
  auto && tmp = <expr>;
  if (!await_ready(tmp) &&
      await_suspend(tmp, <resume-function-object>)) {
  }
  return  await_resume(tmp);
}
```

Awaitable: Better await_suspend

(handle synchronous completion)

```
struct awaiter: awaitable_overlapped {
      void* buf;
      size_t size;
      Connection * my;

         …
      bool await_suspend(coroutine_handle<> cb) {
         callback = cb;
         StartThreadpoolIo(conn->io);
         auto error = TcpSocket::Read(my->handle, buf, size, this);
         if (error == ERROR_IO_PENDING)
            return true;

         CancelThreadpoolIo(conn->io);
         if (error == ERROR_SUCCESS)
            return false;

         throw system_error(error);
      }
      int await_resume() {
         if (ec) throw system_error(ec);
         return nBytes;
      }
};
```

STL looks like the machine language macro library of an anally retentive assembly language programmer

Pamela Seymour, Leiden University

N4134: Layered complexity

- Everybody
  - Safe by default, novice friendly

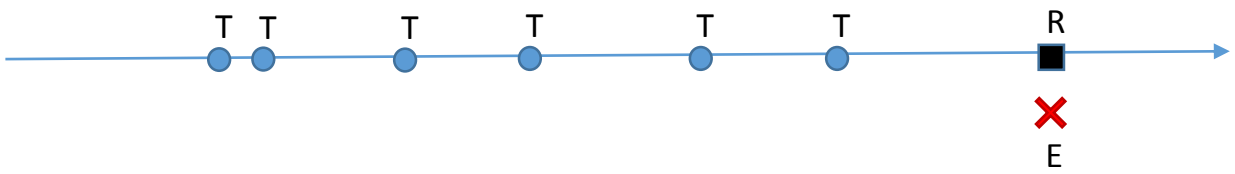
    Use coroutines and awaitables defined by standard library and boost and other high quality libraries
- Power Users
  - Define new awaitables to customize await for their environment using existing coroutine types
- Experts
  - Define new coroutine types

2 x 2 x 2

- Two new keywords
  - **await**
  - **yield**
    syntactic sugar for: **await** $p.yield_value(expr)
- Two new concepts
  - Awaitable
  - Coroutine Promise
- Two library types
  - coroutine_handle
  - coroutine_traits

Coroutine Promise – Concept of an Output Stream

Future<R,E>: (R or E)? Stream<T,R,E>: T* (R or E)?



<promise>.yield_value(T)	`yield expr;`
<promise>.set_result(R)	`return expr;` `implicit return`
<promise>.set_exception(E)	`throw expr;` `unhandled exception`

Could be called from await_suspend or completion callback

Bikeshed		
on_next	emit_value	-
on_complete	return_value	complete(T)
on_error	return_error	complete(E)

Coroutine Frame & Coroutine Promise

coroutine_traits<R,Args…> → CoroutinePromise

Coroutine Return Object

$p.get_return_object()

Coroutine Promise	std::promise<int> $p;
Suspend Context	void * $saved_IP;
Local State	char buf[64 * 1024]; Connection conn; int total; OVERLAPPED $tmp;

```
await $p.initial_suspend();

future<int> tcp_reader(int total)
{
    char buf[64 * 1024];
    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    do
    {
        auto bytesRead = await conn.read(buf, sizeof(buf));
        total -= bytesRead;
    }
    while (total > 0 && bytesRead > 0);
    return total;
}
```

$p.set_result(<expr>?)
$p.set_exception(exception_ptr)

```
await $p.final_suspend();
```

Coroutine Eventual Result

coroutine_traits

```
template <typename R, typename... Args>
R f(Args… args)
```

```
using X = std::coroutine_traits<R, Args…>
```

```
template <typename R, typename... Args>
struct coroutine_traits {

   using promise_type = typename R::promise_type;

   template <typename... Args>
   static auto get_allocator(Args&&…) {
      return std::allocator<char>{};
   }
};
```

coroutine_traits

```
template <typename R, typename... Args>
R f(Args… args)
```

```
using X = std::coroutine_traits<R, Args…>
```

Expression	Note	If not present
X::promise_type	For coroutines with signature above, compiler will place the promise of the specified type on the coroutine frame	R::promise_type
X::get_allocator(args…)	Coroutine will use it to allocate a coroutine frame	std::allocator<char>{}
X::get_return_object_on_allocation_failure()	If present, result of allocate(n) will be checked for nullptr, if nullptr, result of the coroutine will be constructed using X::get_return_object_on_allocation_failture()	assumes that allocate throws (as it should) on failure

N4134 CFAEO

- Coroutine Frame Allocation Elision Optimization
  - An implementation is allowed to elide calls to the allocator's allocate and deallocate functions and use stack memory of the caller instead if the meaning of the program will be unchanged except for the execution of the allocate and deallocate functions.
- Important for async coroutines
  - Allows to break a big async function into many little ones without incurring perf penalty
- Important for generators
  - Makes a generator a zero-overhead abstraction

Coroutine Promise Requirements

Expression	Return type	Note
p.get_return_object()	A type convertible to return type of coroutine	allows connecting Coroutine Promise with Coroutine Return Object
p.set_result([expr])		sets an eventual result of the coroutine. “return <expr>;” or “return;”
p.set_exception(eptr)		Unhandled exception will be forwarded to p.set_exception. If not present exceptions will propagate out of the coroutine even to callers that resumed the coroutine
p.cancellation_requested()	bool	If present, await will have if (cancellation_requested) goto <end> check
p.initial_suspend()	an awaitable type	Suspend after parameter capture
p.final_suspend()	an awaitable type	Suspend prior to destruction

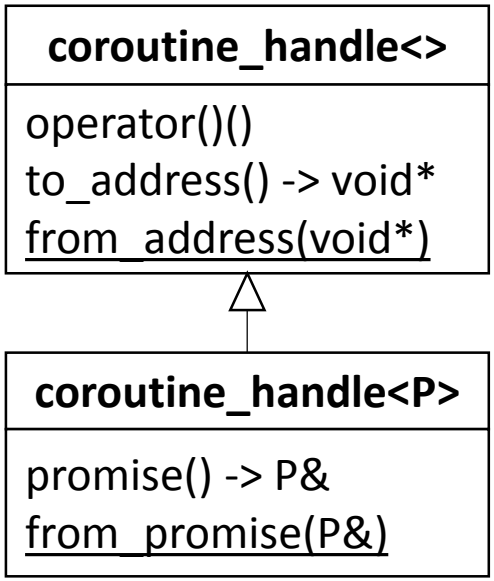
Awaitable: Better await_suspend

```
struct awaiter: awaitable_overlapped {
    …

    bool await_suspend(coroutine_handle<> cb) {
        callback = cb;
        StartThreadpoolIo(conn->io);
        auto error = TcpSocket::Read(my->handle, buf, size, this);
        if (error == ERROR_IO_PENDING)
            return true;

        CancelThreadpoolIo(conn->io);
        if (error == ERROR_SUCCESS)
            return false;


        throw system_error(error);
    }
    …
};
```

Awaitable: Better await_suspend

(preparing to eliminate exceptions)

```
struct awaiter: awaitable_overlapped {
    …
    template <typename Promise>
    bool await_suspend(coroutine_handle<Promise> cb) {
        callback = cb;
        StartThreadpoolIo(conn->io);
        auto error = TcpSocket::Read(my->handle, buf, size, this);
        if (error == ERROR_IO_PENDING)
            return true;

        CancelThreadpoolIo(conn->io);
        if (error == ERROR_SUCCESS)
            return false;


        throw system_error(error);
    }
    …
};
```



Awaitable: Better await_suspend

(propagate exception straight into a coroutine promise)

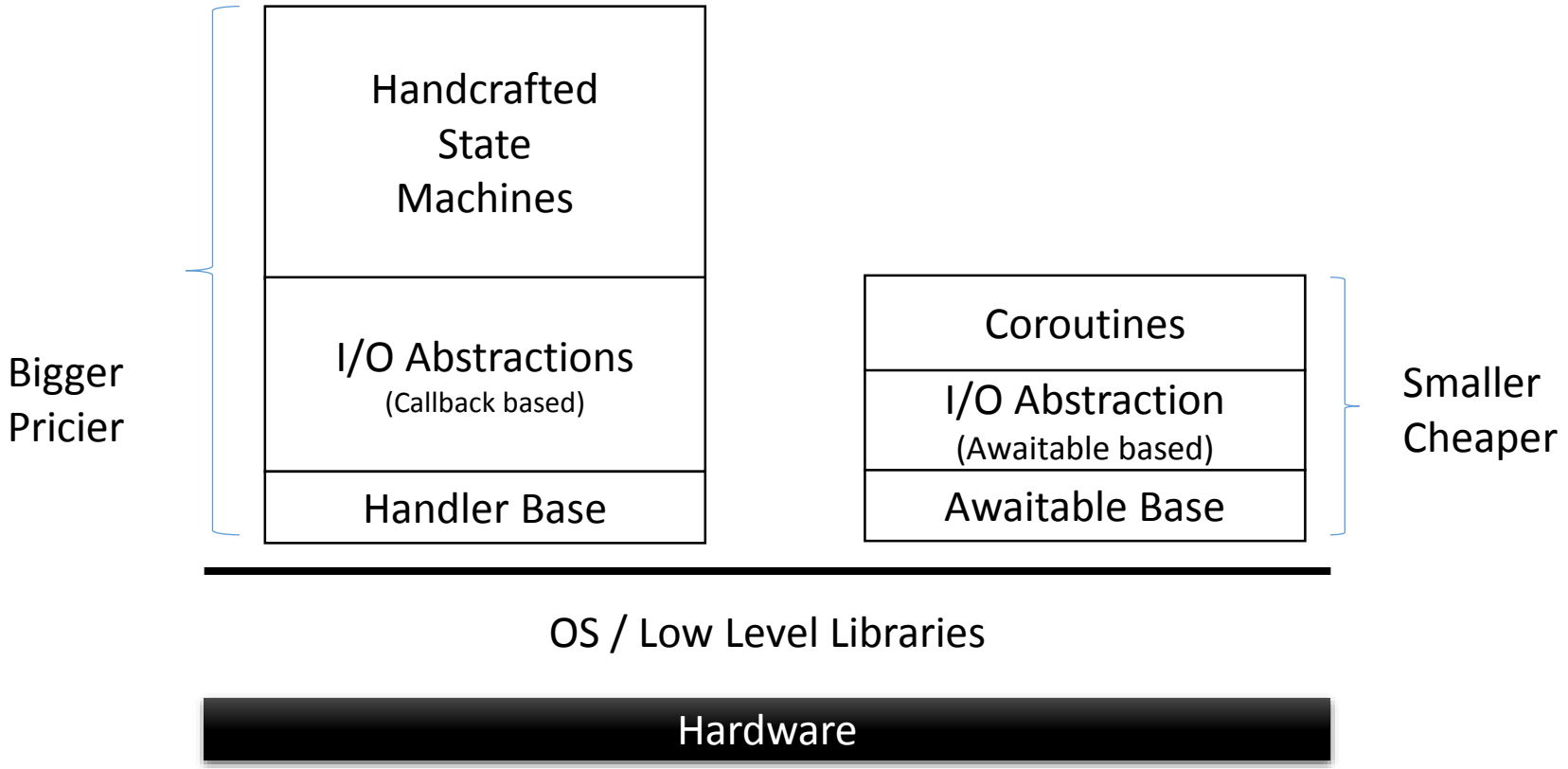
```
struct awaiter: awaitable_overlapped {
    …
    template <typename Promise>
    bool await_suspend(coroutine_handle<Promise> cb) {
        callback = cb;
        StartThreadpoolIo(conn->io);
        auto error = TcpSocket::Read(my->handle, buf, size, this);
        if (error == ERROR_IO_PENDING)
            return true;

        CancelThreadpoolIo(conn->io);
        if (error == ERROR_SUCCESS)
            return false;
        cb.promise().set_exception(
                            make_exception_ptr(system_error(error)));
        return false;
    }
    …
};
```

`await` <expr>

If await_suspend return type is not void, then

```
{
  auto && tmp = <expr>;
  if (!await_ready(tmp) &&
      await_suspend(tmp, <resume-function-object>)) {
  }
  if(<promise>.cancellation_requested()) goto <end>;
  return  await_resume(tmp);
}
```

Coroutines are closer to the metal



Consuming Async Stream

```
future<int> Sum(AsyncStream<int> & input)
{
    int sum = 0;
    for await(v: input)
        sum += v;
    return sum;
}
```

for await (*for-range-declaration : expression*) *statement*

```
{
    auto && __range = range-init;
    for ( auto __begin = await (begin-expr),
          __end = end-expr;
          __begin != __end;
          await ++__begin )
    {
        for-range-declaration = *__begin;
        statement
    }
}
```

N4134 dimensions

	One	Many
Sync	T / Expected<T>	Iterable<T>
Async	Future<T>	AsyncIterable<T>

N4134 can work as a consumer and/or producer for all cases in the table above

N4134: Generic Abstraction

```
M<T> f()
{
    auto x = await f1();
    auto y = await f2();
    return g(x,y);
}
```

Where
- f1: () → M'<X>
- f2: () → M''<Y>
- g: (X,Y) → T
- await: M*<T> → T
- return: T → M<T>

await: unwraps a value from a container M*<T>
return: puts a value back into a container M<T>

Future<T>: container of T, unwrapping strips temporal aspect
optional<T>: container of T, unwrapping strips "not there aspect"
expected<T>: container of T, unwrapping strips "or an error aspect"
std::future<T>: unwrapping strips temporal and may have error aspects