Asynchronous I/O With boost.asio

Avishay Orpaz


avishorp@gmail.com


@avishorp


https://github.com/avishorp

SO, You want to make some I/O....

SO, You want to make some I/O....

That's pretty easy:

```
//Create socket
socket_desc = socket(AF_INET , SOCK_STREAM , 0);

// Bind it
bind(socket_desc,(struct sockaddr *)&server , sizeof(server))

//Listen
listen(socket_desc , 3);

//accept connection from an incoming client
client_sock = accept(socket_desc, (struct sockaddr *)&client,
(socklen_t*)&c);
```

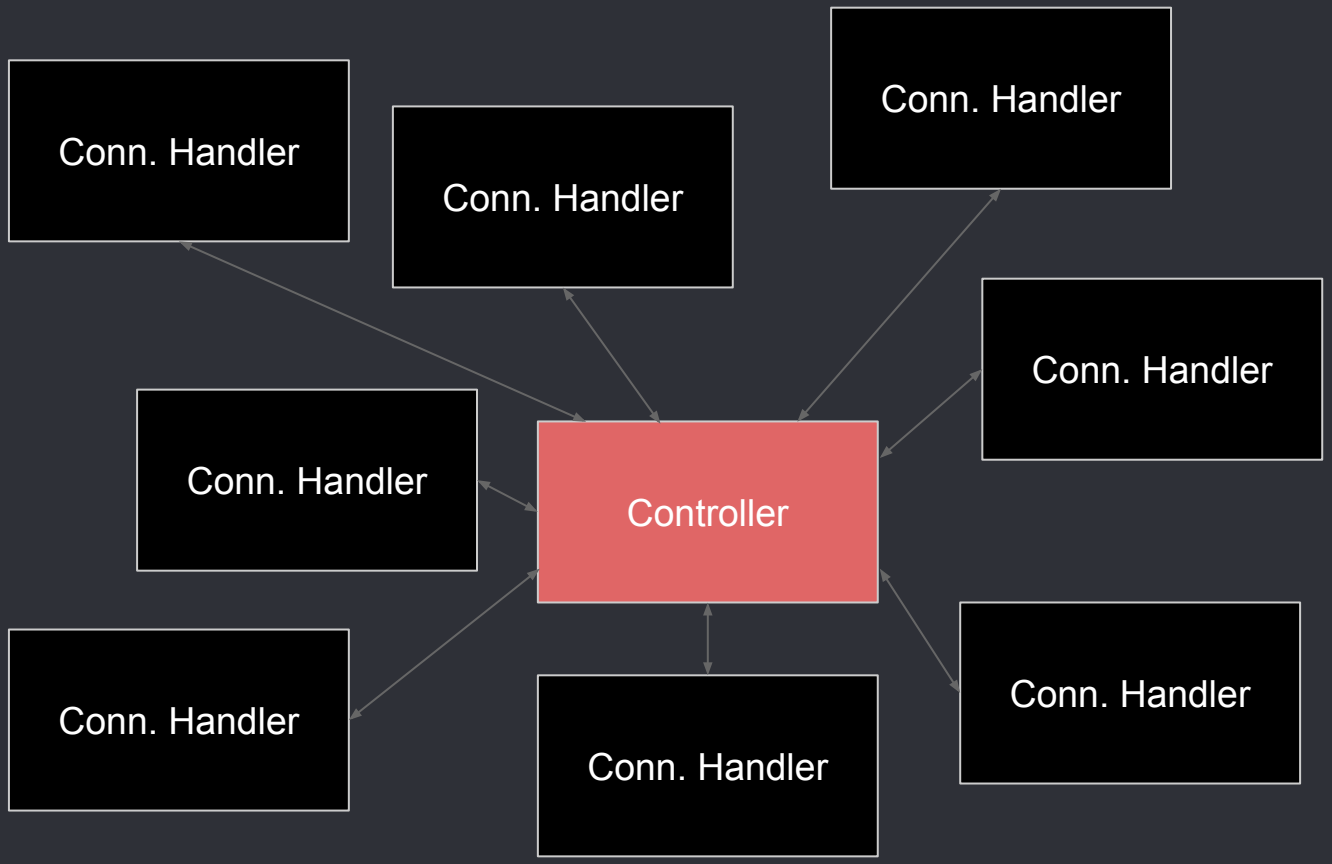
SO, You want to make some I/O....

That's pretty easy:

```
//Create socket
socket_desc = socket(AF_INET , SOCK_STREAM , 0);

// Bind it
bind(socket_desc,(struct sockaddr *)&server , sizeof(server))

//Listen
listen(socket_desc , 3);

//accept connection from an incoming client
client_sock = accept(socket_desc, (struct sockaddr *)&client,
(socklen_t*)&c);
```

But it's blocking :(

Not a problem - let's put it in a thread



That's fine for small number of connections, but does it scale?

Threads do not scale well

- Lots of resource for thread that do nothing most of the time
- Every service requires a full context switch
- Thread design must be safe and reentrant

Warning!

The following slide is *not recommended*

- For those who are allergic to garbage collection
- For those who get stressed without destructors
- For those who are obsessive with pre-compiling their code

Asynchronous I/O in Javascript

```
fs.readFile("myfile.txt", function(data) {
    doSomething(data)
})
```

- When this function gets executed, it starts the I/O operation, then queues a *completion handler*.
- The process is then release to do other things
- When there's nothing else to do, and the I/O completes, the handler will get invoked.

Proactor is a software design pattern for event handling in which long running activities are running in an asynchronous part. A completion handler is called after the asynchronous part has terminated.

Wikipedia, https://en.wikipedia.org/wiki/Proactor_pattern

Enter boost.asio

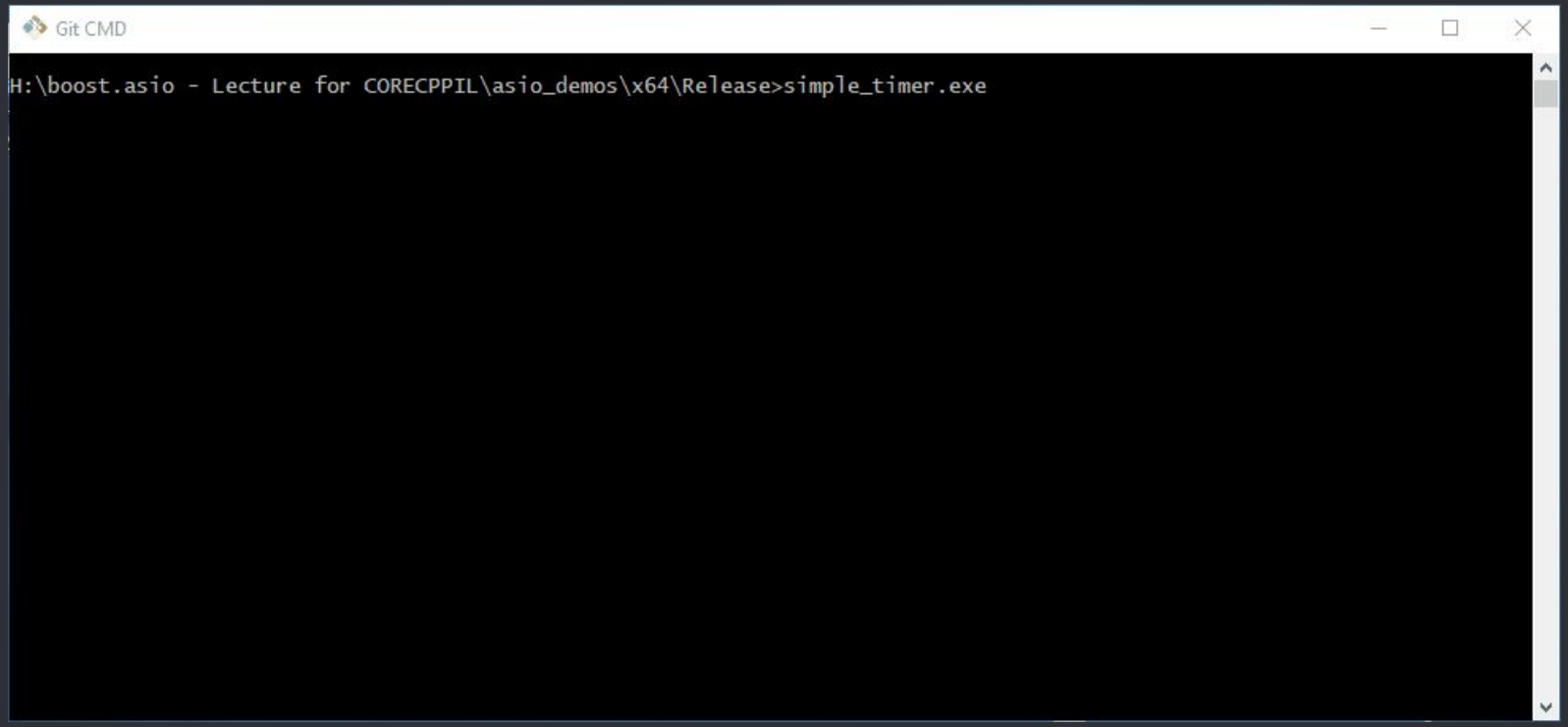
- Written by Christopher Kohlhoff
- Part of boost since 2005
- Provides infrastructure for asynchronous I/O with emphasis on networking.
- Extensible for any other kind of I/O
- Handles only *low-level* communication
- There's also a non-boost variant, called simply `asio`

Getting Started

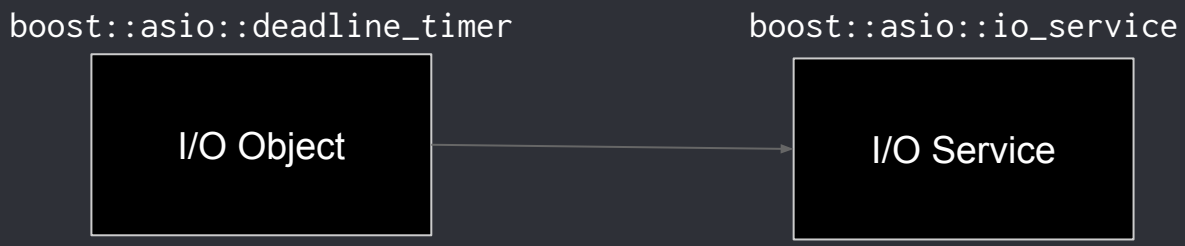
```
int main()
{
    asio::io_service service;

    asio::deadline_timer timer(service, boost::posix_time::seconds(3));

    timer.async_wait([](auto err) {
      std::cout << timestamp << ": Timer expired!\n";
    });

    std::cout << timestamp << ": Calling run\n";
    service.run();
    std::cout << timestamp << ": Done\n";
}
```

simple_timer

Getting Started



ASIO Basics



- Represents an I/O request
- Provides a *completion handler*

- A “main loop”
- Waits for I/O operation to complete
- Invokes the completion handler

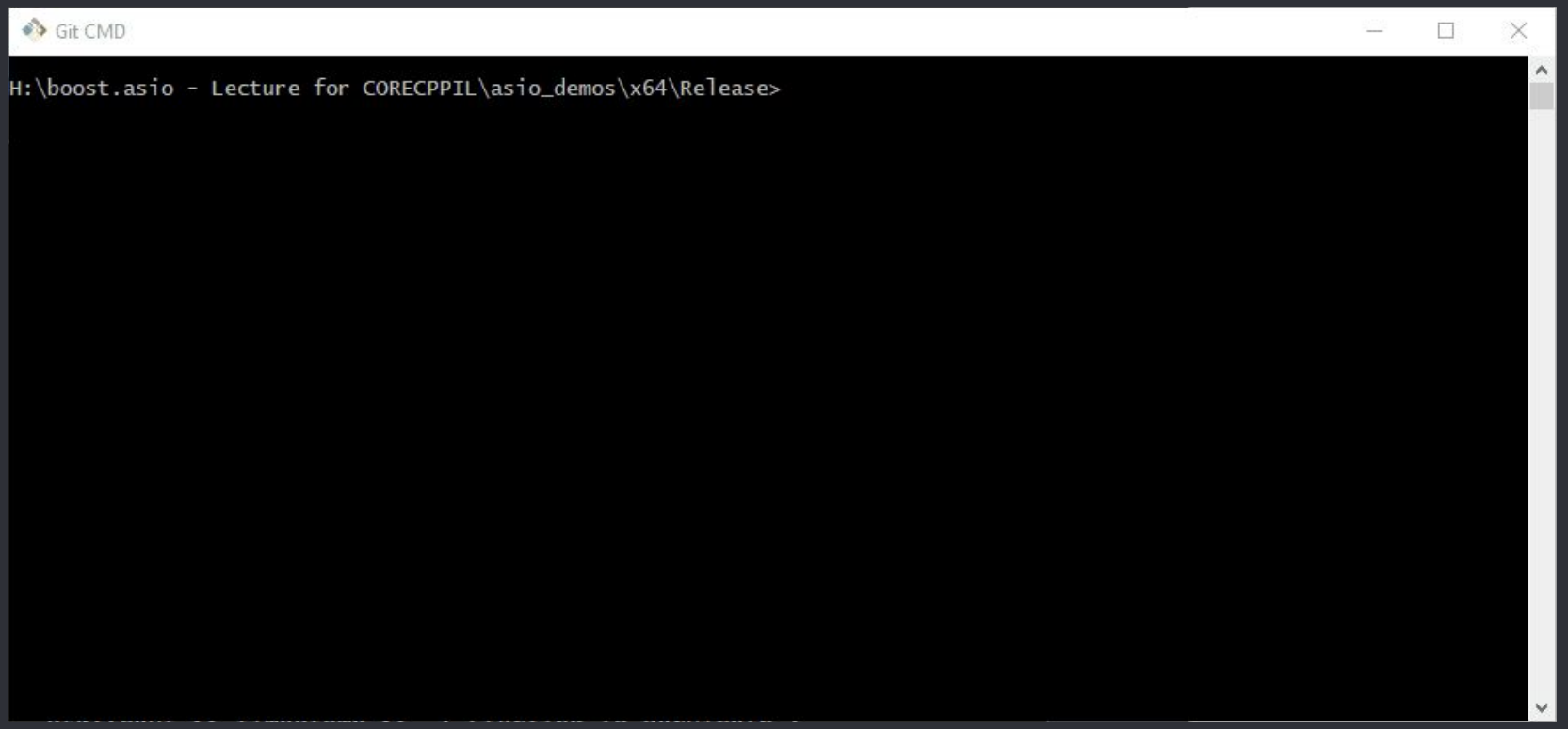
An application may have multiple I/O services, but each I/O object is attached to one I/O service exactly.

Completion Order

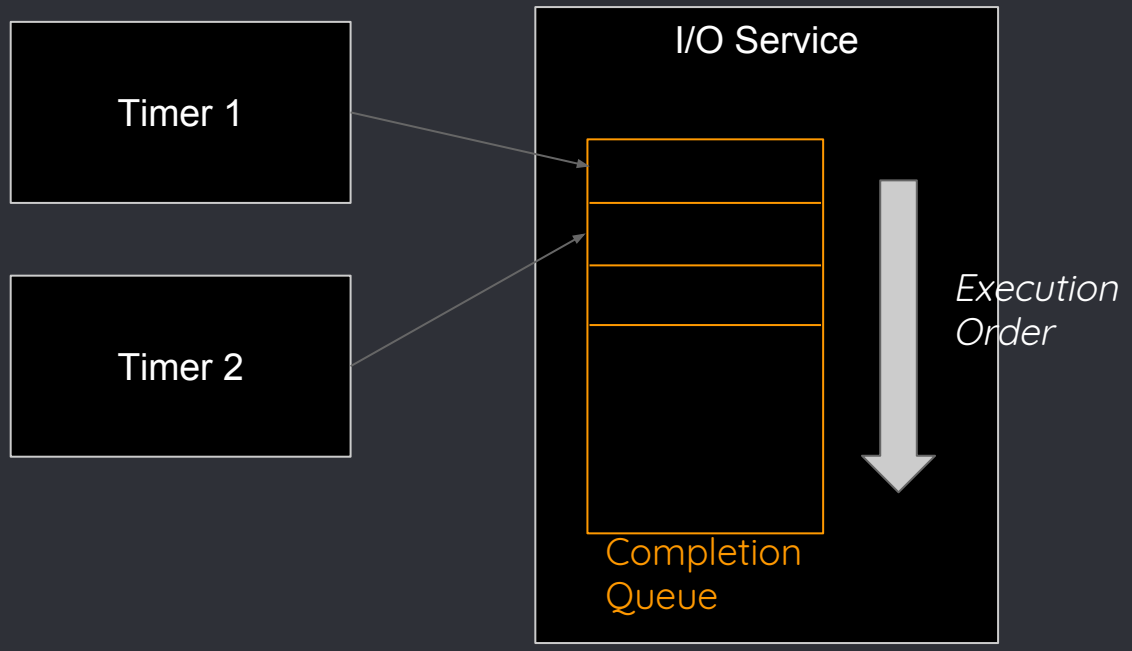
```
int main()
{
    asio::io_service service;

    asio::deadline_timer timer1(service, boost::posix_time::seconds(3));
    asio::deadline_timer timer2(service, boost::posix_time::seconds(3));

    timer1.async_wait([](auto err) {
      std::cout << timestamp << ": Timer 1 expired!\n";
    });

    timer2.async_wait([](auto err) {
      std::cout << timestamp << ": Timer 2 expired!\n";
    });

    std::thread main_loop([&]() {
      std::cout << timestamp << ": Starting io_service\n";
      service.run();
    });
    main_loop.join();
}
```

single_thread

Completion Order



Completion Order



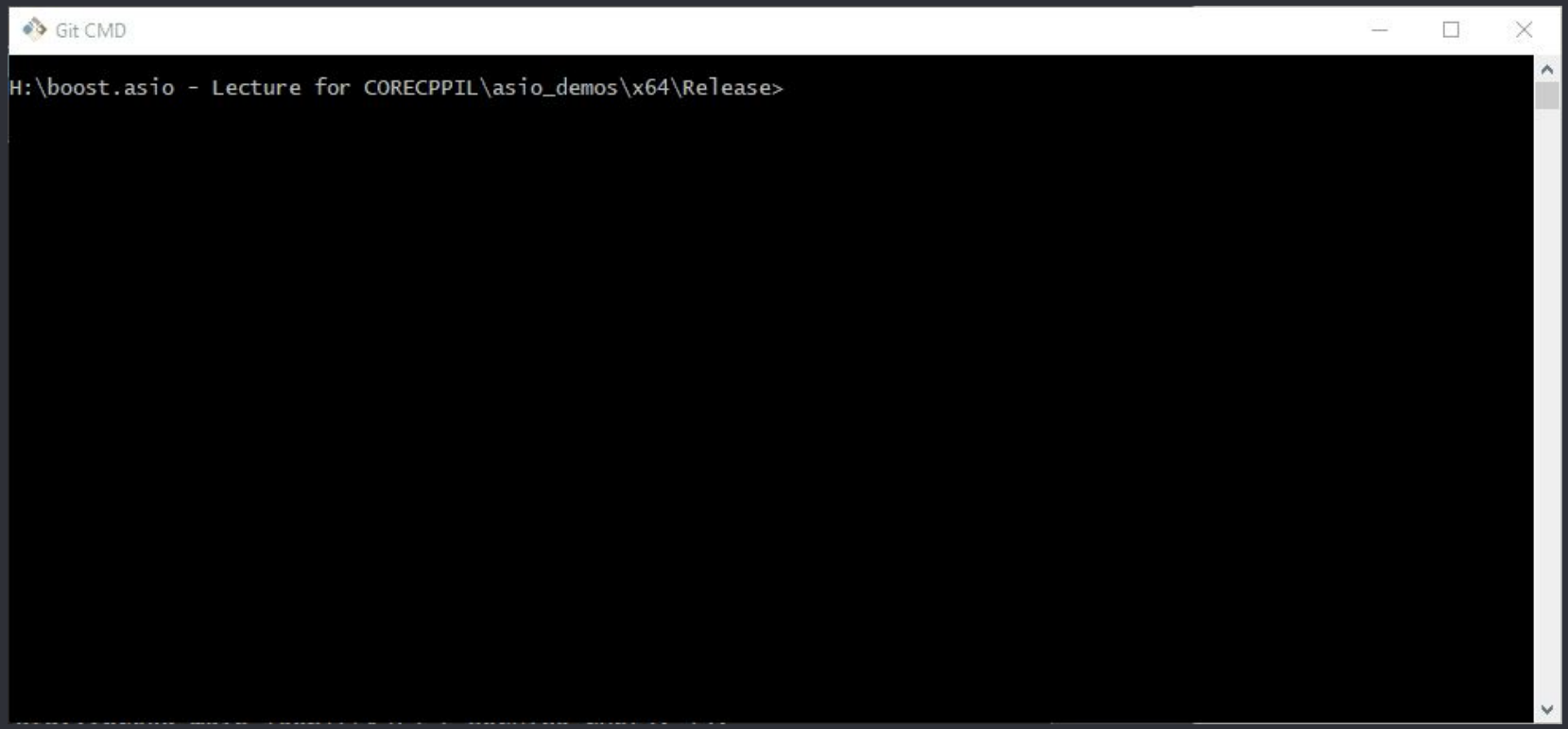
The I/O service picks a completion handler from the queue and executes it.

Multiple Threads

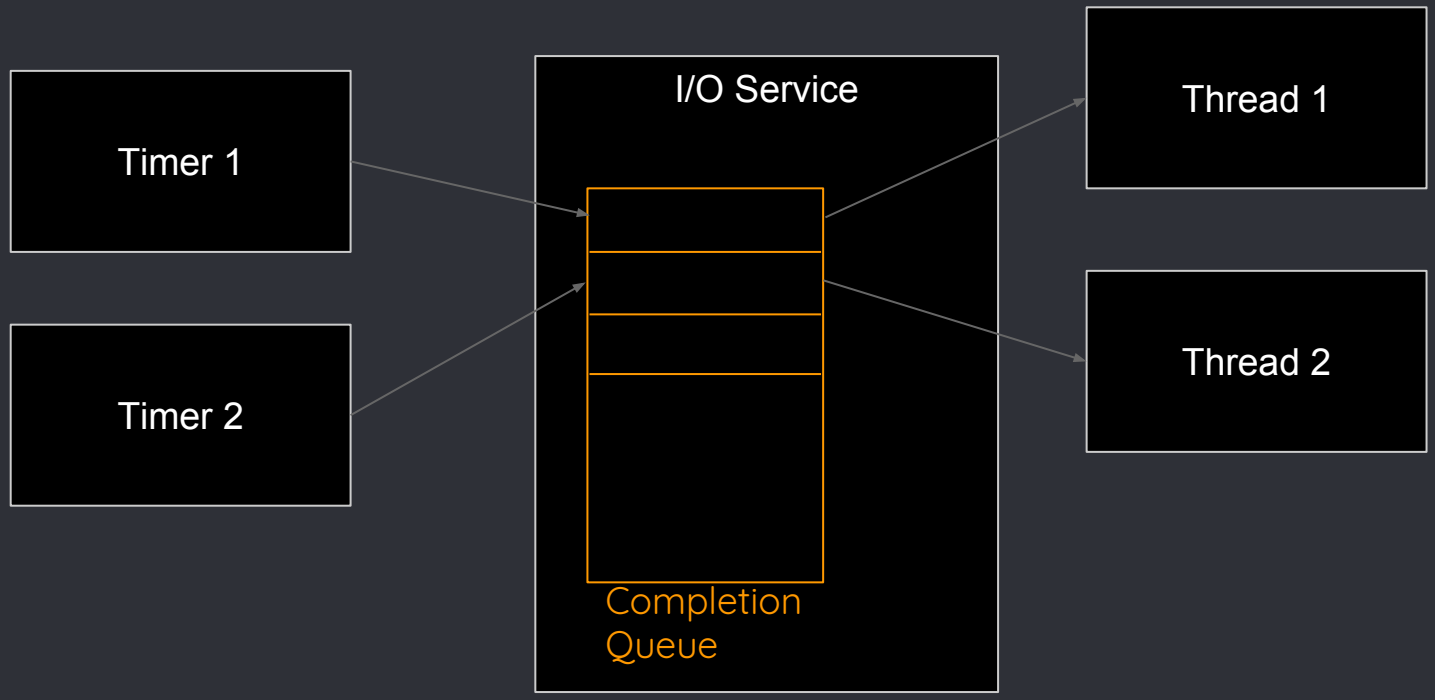
```
int main()
{
    asio::io_service service;

    asio::deadline_timer timer1(service, boost::posix_time::seconds(3));
    asio::deadline_timer timer2(service, boost::posix_time::seconds(3));

    timer1.async_wait([](auto err) {
      std::cout << timestamp << ": Timer 1 expired!\n";
    });

    timer2.async_wait([](auto err) {
      std::cout << timestamp << ": Timer 2 expired!\n";
    });

    // Invoke 2 threads for processing completion handlers
    std::thread main_loop1([&]() { service.run(); });
    std::thread main_loop2([&]() { service.run(); });

    main_loop1.join();
    main_loop2.join();
}
```

multiple_threads

Multiple Threads



Completion Order



Multiple threads can be attached to an I/O service to create a thread pool. Whenever a handler is ready, one of the threads will pick it up and execute it.

Strands

```
int main()
{
    asio::io_service service;
    asio::io_service::strand strand(service);

    asio::deadline_timer timer1(service, boost::posix_time::seconds(3));
    asio::deadline_timer timer2(service, boost::posix_time::seconds(3));

    timer1.async_wait(strand.wrap([](auto err) {
      std::cout << timestamp << ": Timer 1 expired!\n";
    }));

    timer2.async_wait(strand.wrap([](auto err) {
      std::cout << timestamp << ": Timer 2 expired!\n";
    }));

    // Invoke 2 threads for processing completion handlers
    std::thread main_loop1([&]() { service.run(); });
    std::thread main_loop2([&]() { service.run(); });

    main_loop1.join();
    main_loop2.join();
}
```

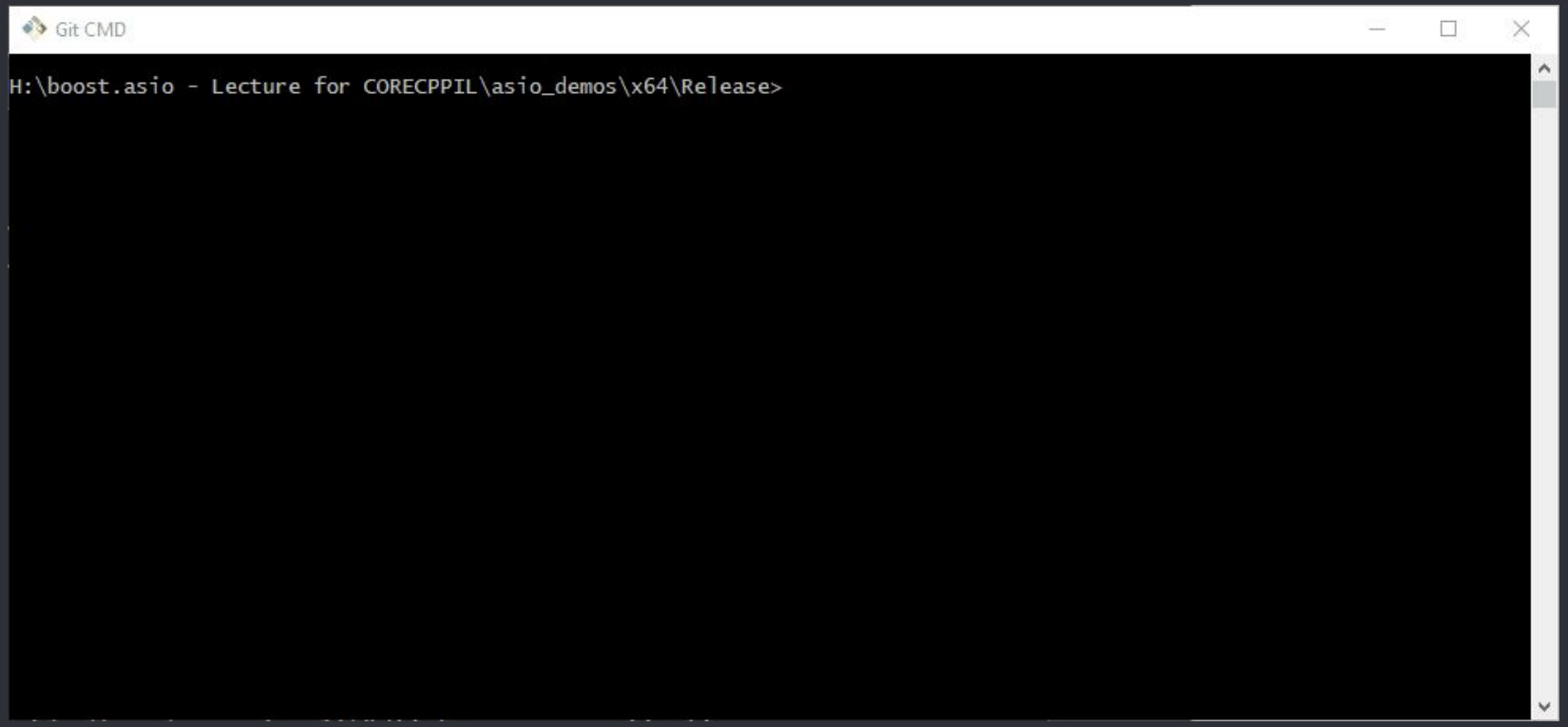
strand

Strands

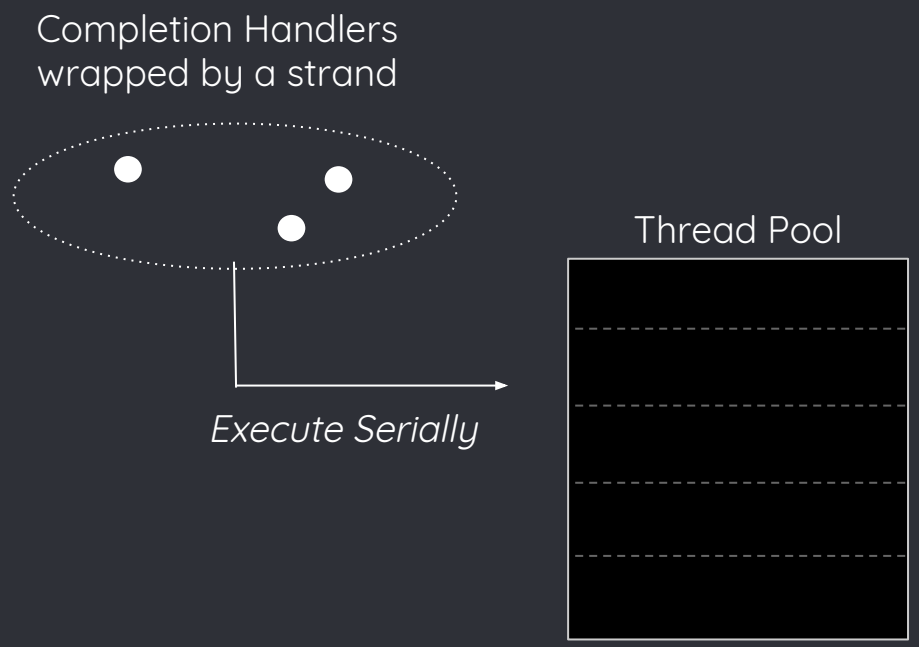
```
int main()
{
    asio::io_service service;
    asio::io_service::strand strand(service);

    asio::deadline_timer timer1(service, boost::posix_time::seconds(3));
    asio::deadline_timer timer2(service, boost::posix_time::seconds(3));

    timer1.async_wait(strand.wrap([](auto err) {
      std::cout << timestamp << ": Timer 1 expired!\n";
    }));

    timer2.async_wait(strand.wrap([](auto err) {
      std::cout << timestamp << ": Timer 2 expired!\n";
    }));

    // Invoke 2 threads for processing completion handlers
    std::thread main_loop1([&]() { service.run(); });
    std::thread main_loop2([&]() { service.run(); });

    main_loop1.join();
    main_loop2.join();
}
```

strand

Strands


Git CMD
H:\boost.asio - Lecture for CORECPPIL\asio_demos\x64\Release>

Strands



Strand is a synchronization mechanism. Only one compl. Handler, wrapped by a strand will be executed in any given time.

Networking with boost::asio

Networking with boost::asio

- Boost::asio is first and foremost a *networking library*.
- Provides abstractions for common network related objects:
  - Sockets
  - Addresses
  - Name resolution
  - Buffers
- Also, built-in serial port support

Example: Asynchronous HTTP GET

```
int main()
{
    tcp::resolver::query q{ "theboostcpplibraries.com", "80" };
    resolv.async_resolve(q, resolve_handler);

    ioservice.run();
}
```

First, we have to resolve the address. We have a `boost::asio::tcp::resolver` object to handle that.

Example: Asynchronous HTTP GET

```
void resolve_handler(const boost::system::error_code &ec,
    tcp::resolver::iterator it)
{
    if (!ec)
      tcp_socket.async_connect(*it, connect_handler);
}
```

When the address is resolved, the `resolve_handler` function will be executed.

If it completed without errors, we can try to connect using a `boost::asio::tcp_scoket`

Example: Asynchronous HTTP GET

```
void connect_handler(const boost::system::error_code &ec)
{
    if (!ec)
    {
      std::string r =
           "GET / HTTP/1.1\r\nHost: theboostcpplibraries.com\r\n\r\n";
      write(tcp_socket, buffer(r));
      tcp_socket.async_read_some(buffer(bytes), read_handler);
    }
}
```

The `connect_handler` function will be called when the connection is ready.

We write the request (synchronously) then issue an asynchronous read request.

Example: Asynchronous HTTP GET

```
std::array<char, 4096> bytes;

void connect_handler(const boost::system::error_code &ec)
{
    if (!ec)
    {
      std::string r =
           "GET / HTTP/1.1\r\nHost: theboostcpplibraries.com\r\n\r\n";
      write(tcp_socket, buffer(r));
      tcp_socket.async_read_some(buffer(bytes), read_handler);
    }
}
```

A `boost::asio::buffer` object wraps the actual buffer in memory. It must be valid across the whole scope of the read.

Example: Asynchronous HTTP GET

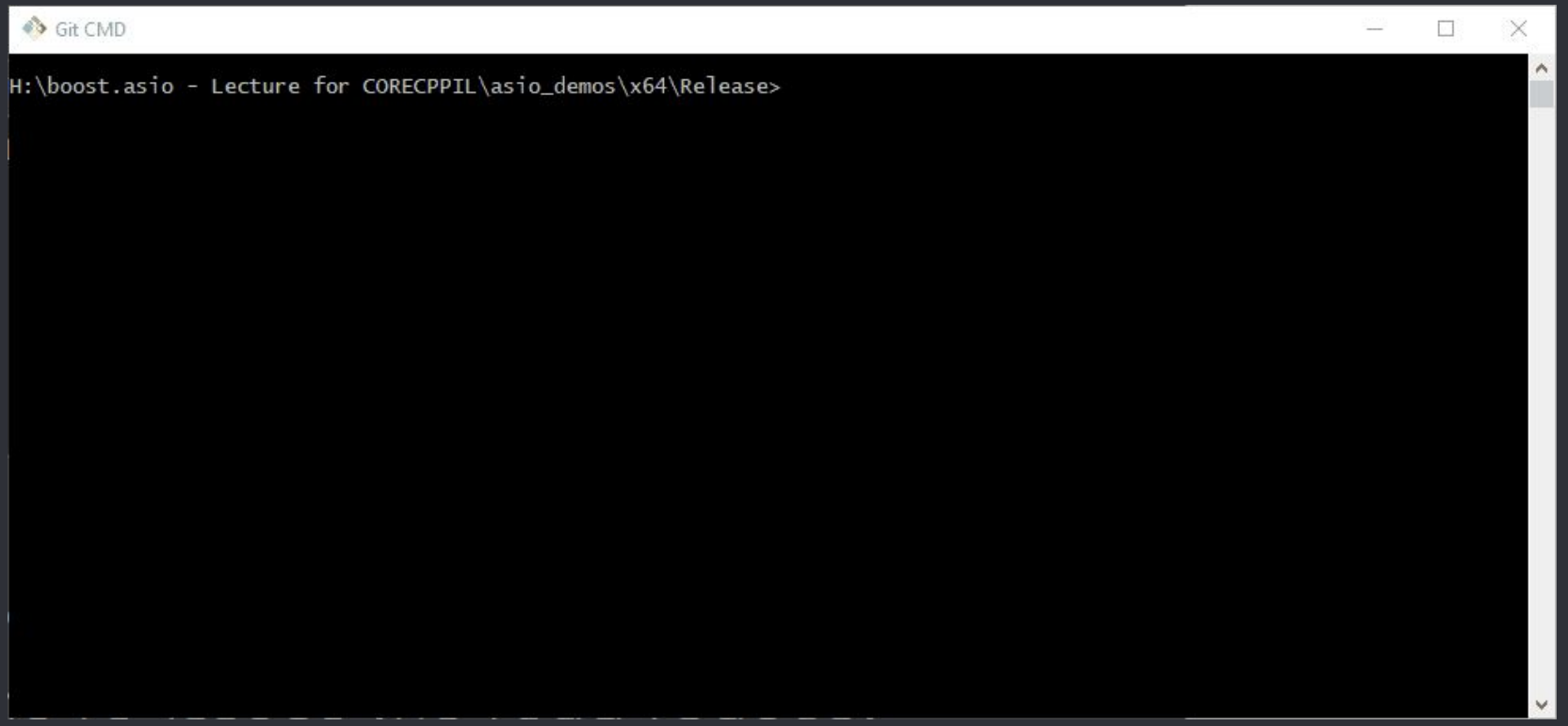
```
void read_handler(const boost::system::error_code &ec,
    std::size_t bytes_transferred)
{
    if (!ec)
    {
      std::cout.write(bytes.data(), bytes_transferred);
      tcp_socket.async_read_some(buffer(bytes), read_handler);
    }
    else
      std::cout << "End of stream" << std::endl;
}
```

This is not a recursion!

The `read_handler` function will be called when data has arrived.

It the re-issues the read request until no more data is available.

Example: Asynchronous HTTP GET

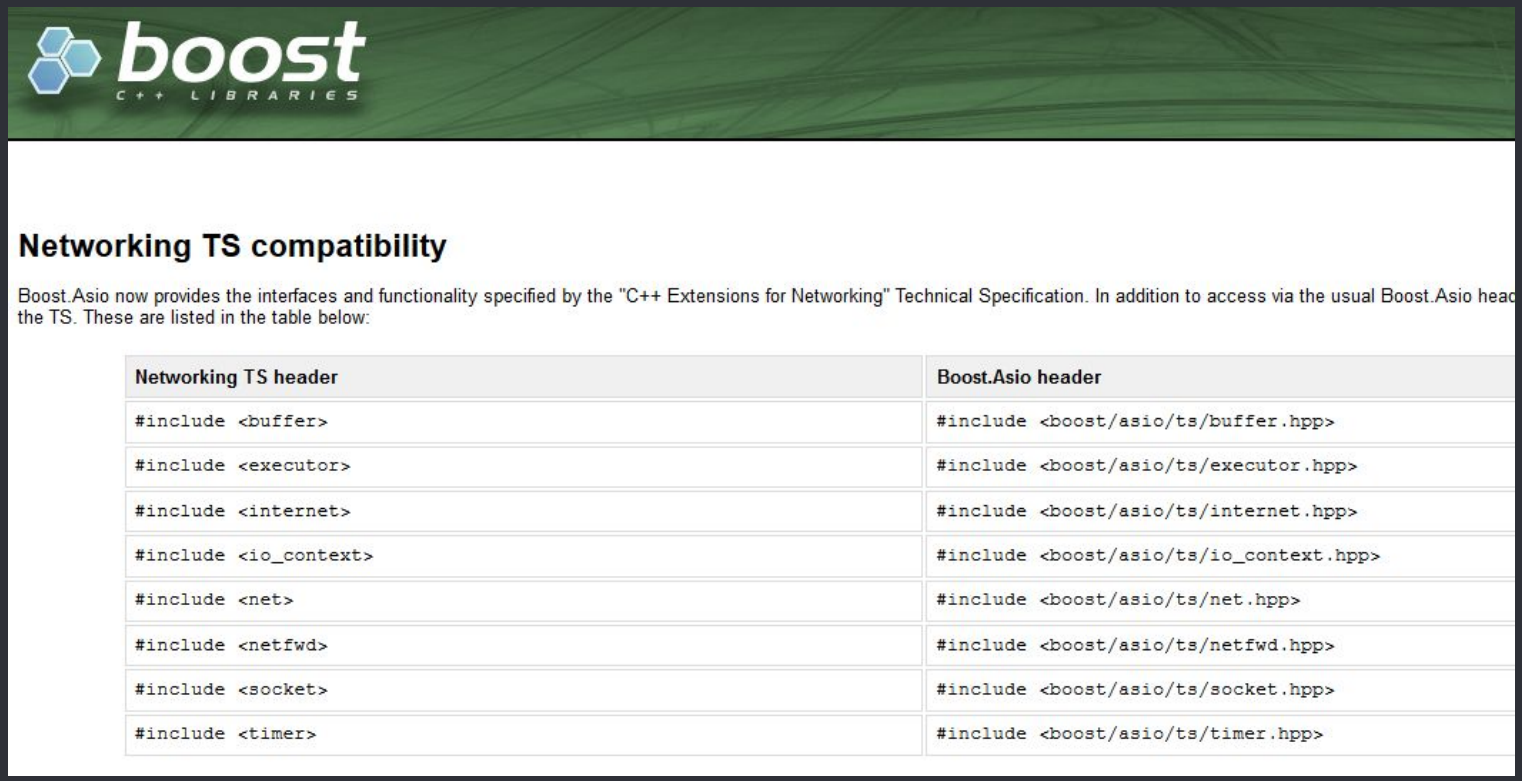


Boost.asio and Networking-TS

- Networking TS is a broad scope endeavor to standardize networking in C++
- It has both sync & async semantics
- Async is heavily based on boost::asio
- It also borrows concepts such as buffers
- Change in names (So we have something new to learn)

Boost.asio and Networking-TS

- Since boost 1.66.0, compatibility headers are provided
- See here



boost C++ LIBRARIES

Networking TS compatibility

Boost.Asio now provides the interfaces and functionality specified by the "C++ Extensions for Networking" Technical Specification. In addition to access via the usual Boost.Asio heac
the TS. These are listed in the table below:

Networking TS header	Boost.Asio header
#include <buffer>	#include <boost/asio/ts/buffer.hpp>
#include <executor>	#include <boost/asio/ts/executor.hpp>
#include <internet>	#include <boost/asio/ts/internet.hpp>
#include <io_context>	#include <boost/asio/ts/io_context.hpp>
#include <net>	#include <boost/asio/ts/net.hpp>
#include <netfwd>	#include <boost/asio/ts/netfwd.hpp>
#include <socket>	#include <boost/asio/ts/socket.hpp>
#include <timer>	#include <boost/asio/ts/timer.hpp>

Asynchronous file I/O

It is possible to do asynchronous file I/O with boost::asio

It is possible to do asynchronous file I/O with boost::asio

Asynchronous File I/O

- Currently, File I/O is not supported in a platform independent manner.
- Windows uses OVERLAPPED I/O requests.
- Posix is a mess.

Asynchronous File I/O In Windows

```
HANDLE file_handle = CreateFileA(".", FILE_LIST_DIRECTORY,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE, NULL,
    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED,
    NULL);

OVERLAPPED overlapped;
overlapped.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
ReadDirectoryChangesW(file_handle, buffer, sizeof(buffer), FALSE,
     FILE_NOTIFY_CHANGE_FILE_NAME, &transferred, &overlapped, NULL);

windows::object_handle obj_handle{ioservice, overlapped.hEvent};

obj_handle.async_wait([&buffer, &overlapped](const error_code &ec) {
     ...
     GetOverlappedResult(overlapped.hEvent, &overlapped, &transferred,
     FALSE);
     ...
     }
 });

ioservice.run();
```

Asynchronous File I/O In Windows

```
HANDLE file_handle = CreateFileA(".", FILE_LIST_DIRECTORY,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE, NULL,
    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED,
    NULL);

OVERLAPPED overlapped;
overlapped.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
ReadDirectoryChangesW(file_handle, buffer, sizeof(buffer), FALSE,
     FILE_NOTIFY_CHANGE_FILE_NAME, &transferred, &overlapped, NULL);

windows::object_handle obj_handle{ioservice, overlapped.hEvent};

obj_handle.async_wait([&buffer, &overlapped](const error_code &ec) {
     ...
     GetOverlappedResult(overlapped.hEvent, &overlapped, &transferred,
     FALSE);
     ...
     }
 });

ioservice.run();
```

Create a file with FILE_FLAG_OVERLAPPED

Asynchronous File I/O In Windows

```
HANDLE file_handle = CreateFileA(".", FILE_LIST_DIRECTORY,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE, NULL,
    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED,
    NULL);

OVERLAPPED overlapped;
overlapped.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
ReadDirectoryChangesW(file_handle, buffer, sizeof(buffer), FALSE,
     FILE_NOTIFY_CHANGE_FILE_NAME, &transferred, &overlapped, NULL);

windows::object_handle obj_handle{ioservice, overlapped.hEvent};

obj_handle.async_wait([&buffer, &overlapped](const error_code &ec) {
     ...
     GetOverlappedResult(overlapped.hEvent, &overlapped, &transferred,
     FALSE);
     ...
     }
 });

ioservice.run();
```

Issue an overlapped I/O action, providing an `OVERLAPPED` structure and an event.

Asynchronous File I/O In Windows

```
HANDLE file_handle = CreateFileA(".", FILE_LIST_DIRECTORY,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE, NULL,
    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED,
    NULL);

OVERLAPPED overlapped;
overlapped.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
ReadDirectoryChangesW(file_handle, buffer, sizeof(buffer), FALSE,
     FILE_NOTIFY_CHANGE_FILE_NAME, &transferred, &overlapped, NULL);

windows::object_handle obj_handle{ioservice, overlapped.hEvent};

obj_handle.async_wait([&buffer, &overlapped](const error_code &ec) {
     ...
     GetOverlappedResult(overlapped.hEvent, &overlapped, &transferred,
     FALSE);
     ...
     }
 });

ioservice.run();
```

Create a `boost::asio::windows::object_handle` object that binds the I/O service to the event handle

Asynchronous File I/O In Windows

```
HANDLE file_handle = CreateFileA(".", FILE_LIST_DIRECTORY,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE, NULL,
    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED,
    NULL);

OVERLAPPED overlapped;
overlapped.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
ReadDirectoryChangesW(file_handle, buffer, sizeof(buffer), FALSE,
     FILE_NOTIFY_CHANGE_FILE_NAME, &transferred, &overlapped, NULL);

windows::object_handle obj_handle{ioservice, overlapped.hEvent};

obj_handle.async_wait([&buffer, &overlapped](const error_code &ec) {
     ...
     GetOverlappedResult(overlapped.hEvent, &overlapped, &transferred,
     FALSE);
     ...
     }
 });

ioservice.run();
```

Specify a function to receive the result or the error code

Asynchronous File I/O In POSIX

```
io_service ioservice;

 posix::stream_descriptor stream{ioservice, STDOUT_FILENO};
 auto handler = [](const boost::system::error_code&, std::size_t) {
   std::cout << ", world!\n";
 };
 async_write(stream, buffer("Hello"), handler);

 ioservice.run();
```

- The basic type here is `posix::stream_descriptor`.
- It's a wrapper around platform-specific file descriptor
- Provide async stream semantics

Learning More

Asynchronous File I/O

- YouTube, talks by Michael Caisse and others
- Nice, extensive getting started
  https://theboostcpplibraries.com/boost.asio
- Old, but covers things that are not usually covered
  https://www.gamedev.net/blogs/entry/2249317-a-guide-to-getting-started-with-boostasio/
- Boost.asio official documentation
- RTFC