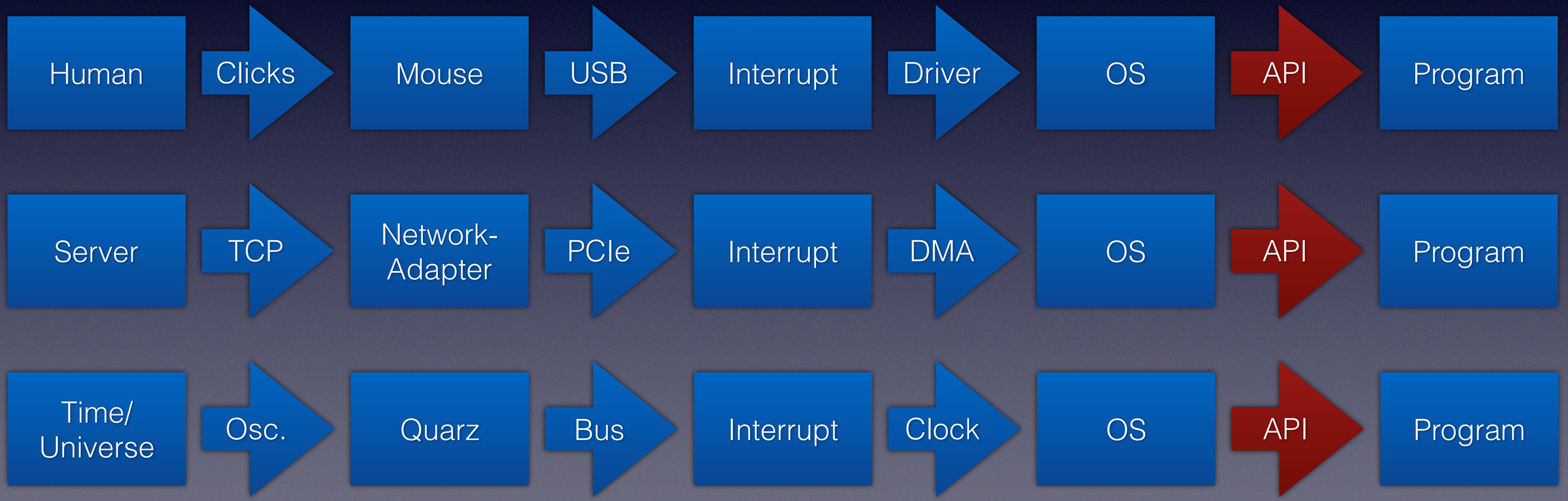
C++20 Coroutines, with Boost ASIO in production: Frightening but awesome

Markus Klemm
www.markusklemm.net

TOC

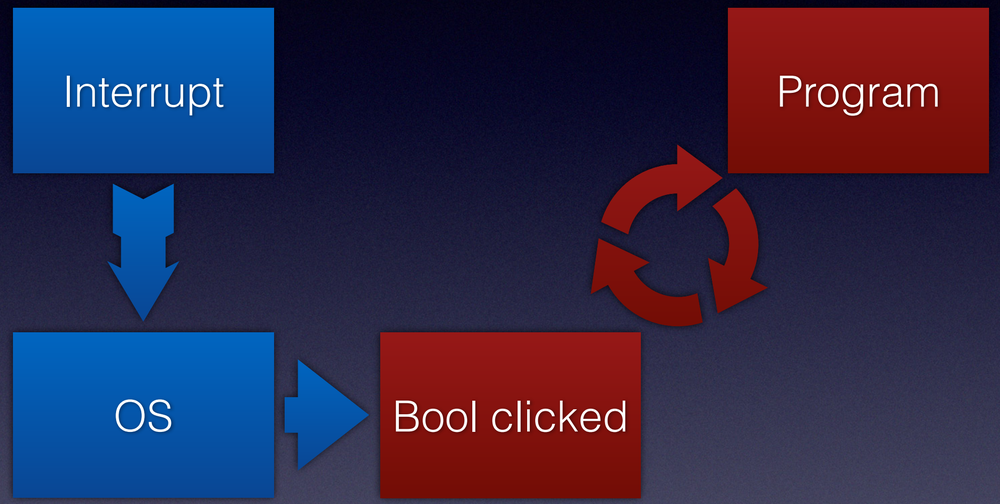
- Little Recap: What is async? Why is async? How is async?
  - We live in a interrupt, trigger based world
  - Concurrency/Multithreading, we are not the same
  - Reactor Pattern

Life is a series of moments



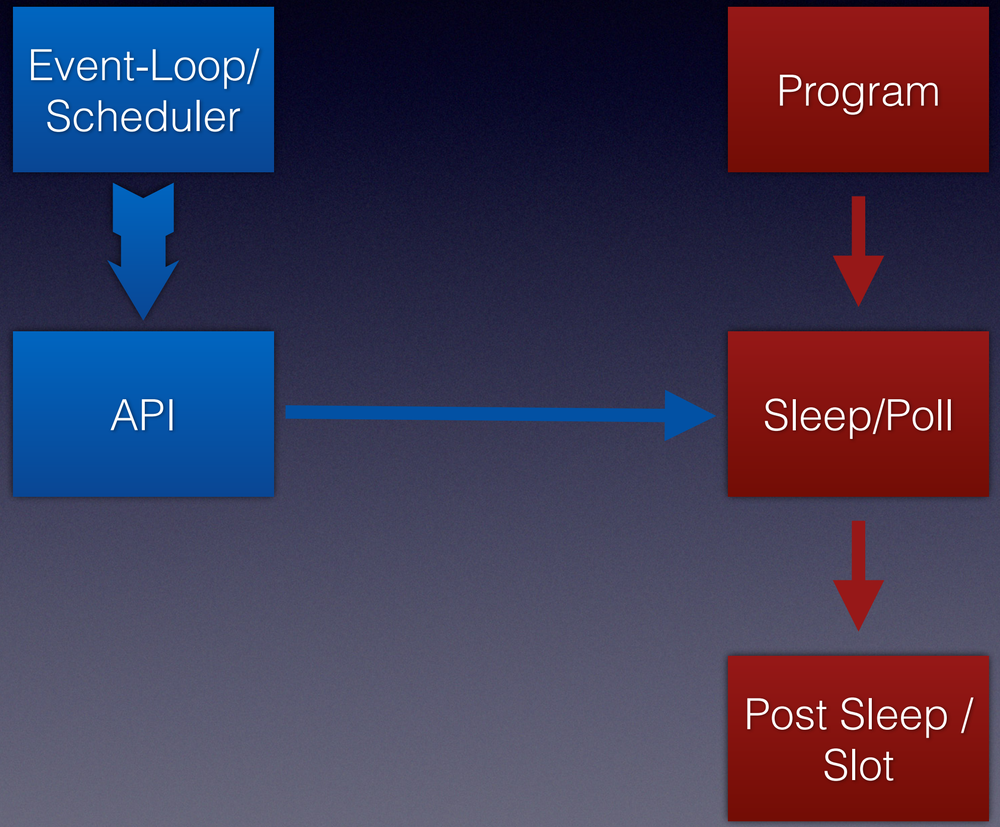
Polling: Busy wait

- while(!button.clicked());
- while(cpu_cycle_counter < 1337000);
- „Are we there yet?“-„No“„Are we there yet?“-„No“„Are we there yet?“-„No“„Are we there yet?“-„No“„Are we there yet?“-„No“
- Ressource waste
- Not working without preemptive multitasking (no embedded, freestanding, bare metal)



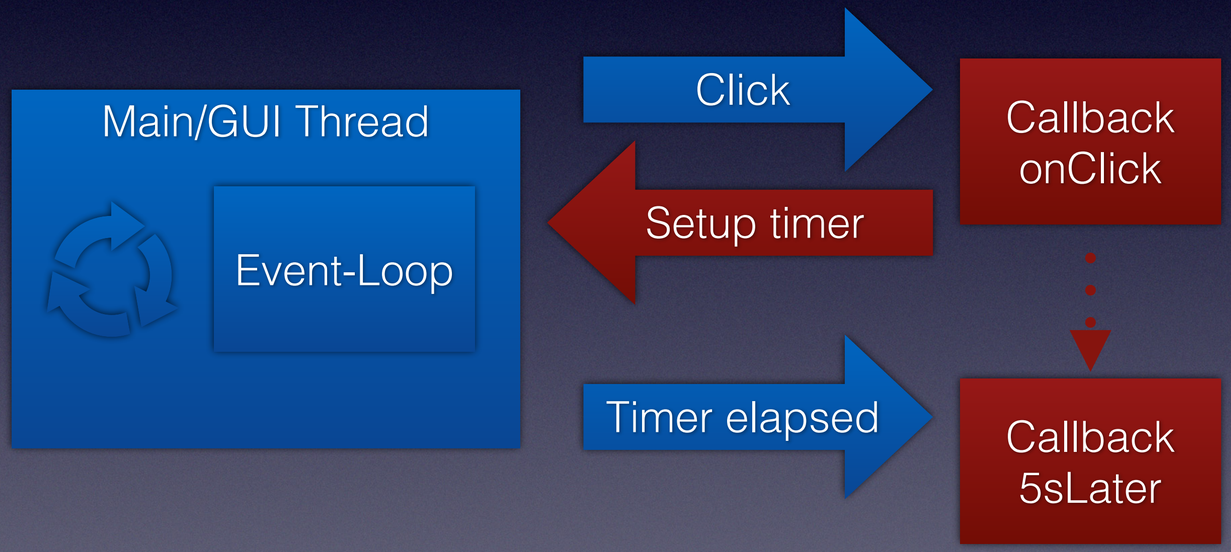
Polling: Blocking API

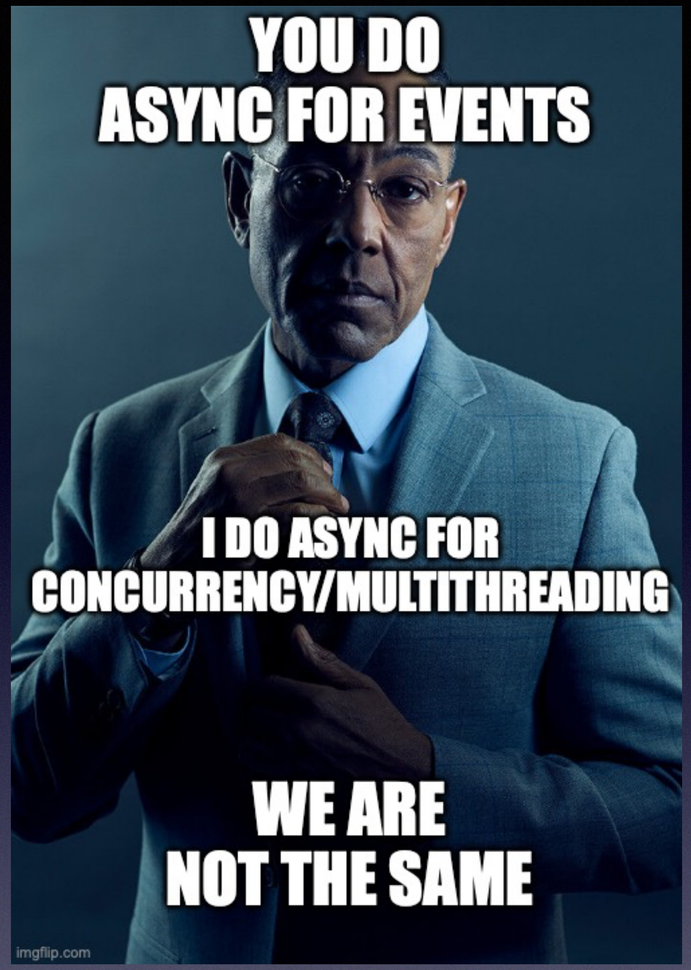
- SDL_PollEvent()
- boost::asio::steady_timer::wait();
- std::this_thread::sleep_for(std::chrono::seconds{2});
- „I go sleep, wake me up when something happens“



Async: Java, Qt, Javascript

- Qt::QCoreApplication::exec()
- Browser JS main thread
- One Thread to rule them all
- No fear of data races, no mutex, no strand needed
- Callback can block main thread



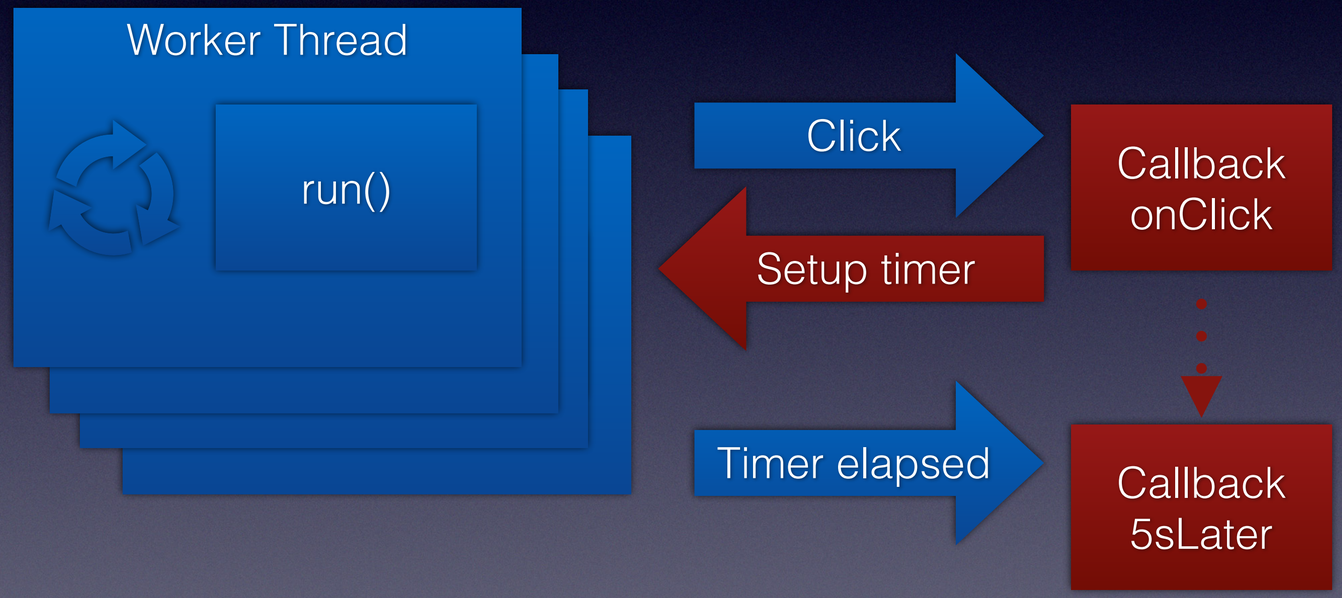


What does this have to do with concurrency?

Proactor pattern

One simple but major change

- Thread[n]: Boost::asio::io_context::run()
- One of the Worker threads will call the callbacks
- Callback only blocks one worker thread
- Is optional: You can just call run() in one thread, might be just the main() thread
- To put simply: userland cooperative multitasking
- Boost can use OS native async facilities, up to transitive from the original interrupt



OK, but was has this to do with coroutines?

Callback-Hell

- „Wake me in 1 second. If this happens: do this, do that, and wake me again, if this happens: do that….“
- Pyramid of doom
- Loops no longer visible
- Lifetime/Scope-Issues

```
void op_a() {
    std::cout << "op_a " << ++session_data_ << "\n";
    timer_.expires_after(std::chrono::seconds{1});
    timer_.async_wait([self = shared_from_this()] (const auto& boost_error){
        if (boost_error) {
            throw std::runtime_error(boost_error.message());
        }
        self->op_b();
        self->timer_.expires_after(std::chrono::seconds{1});
        self->timer_.async_wait([self = self->shared_from_this()] (const auto& boost_error){
            if (boost_error) {
                throw std::runtime_error(boost_error.message());
            }
            self->op_a();
        });
    });
```

Javascript Evolution

- From Callback-Hell to

```
setTimeout( handler: (arg) :void => {
    console.log(`arg1 was => ${arg}`);
    setTimeout( handler: (arg) :void => {
        console.log(`arg2 was => ${arg}`);
        setTimeout( handler: (arg) :void => {
            console.log(`arg3 was => ${arg}`);}, timeout: 1500, arguments: 'funky3');
    }, timeout: 1500, arguments: 'funky2');
}, timeout: 1500, arguments: 'funky1');
```

- Promises aka then.then.then.catch to

```
timersPromises.setTimeout( handler: 3000, timeout: "Promise 1")
    .then((value) => {
    console.log("Then ", value); return timersPromises.setTimeout( handler: 1000, timeout: 'Promise2');})
    .then( onfulfilled: (value) => {
    console.log("Then ", value); return timersPromises.setTimeout( handler: 1000, timeout: 'Promise3');});
```

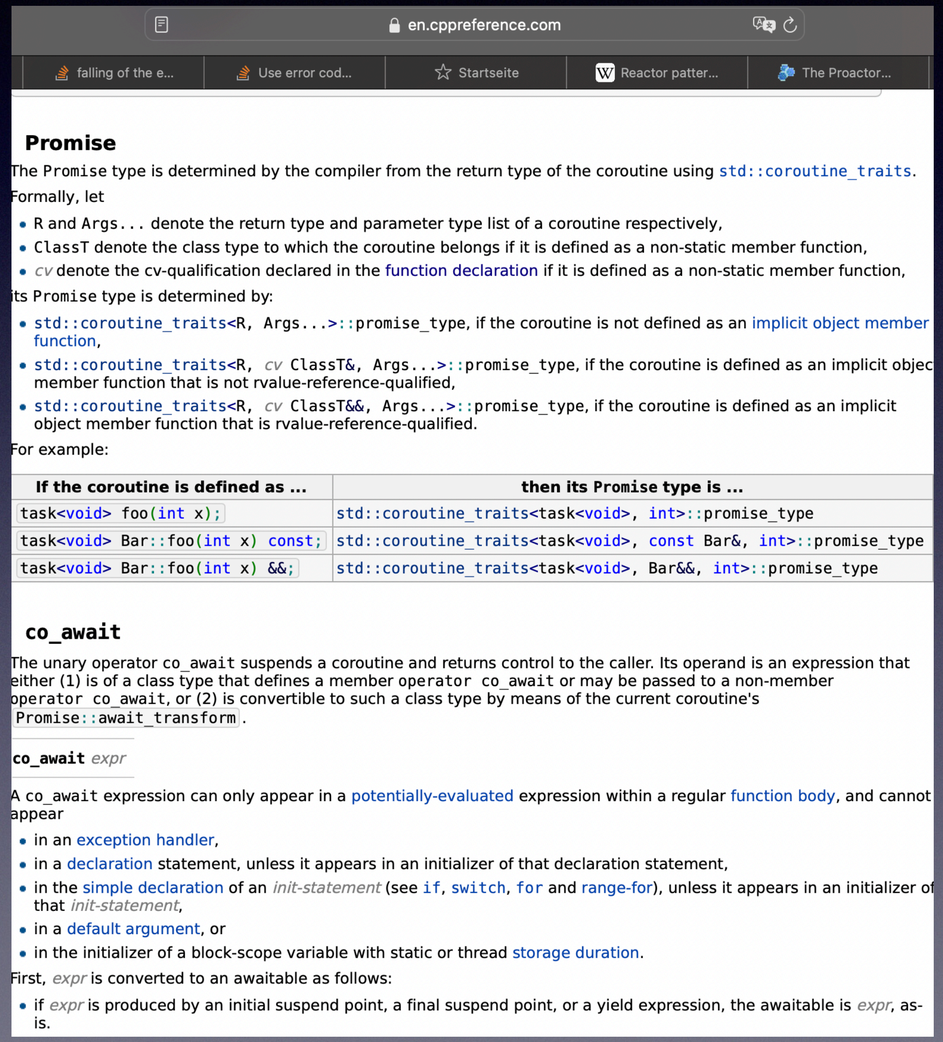
- Async/Await/Coroutines

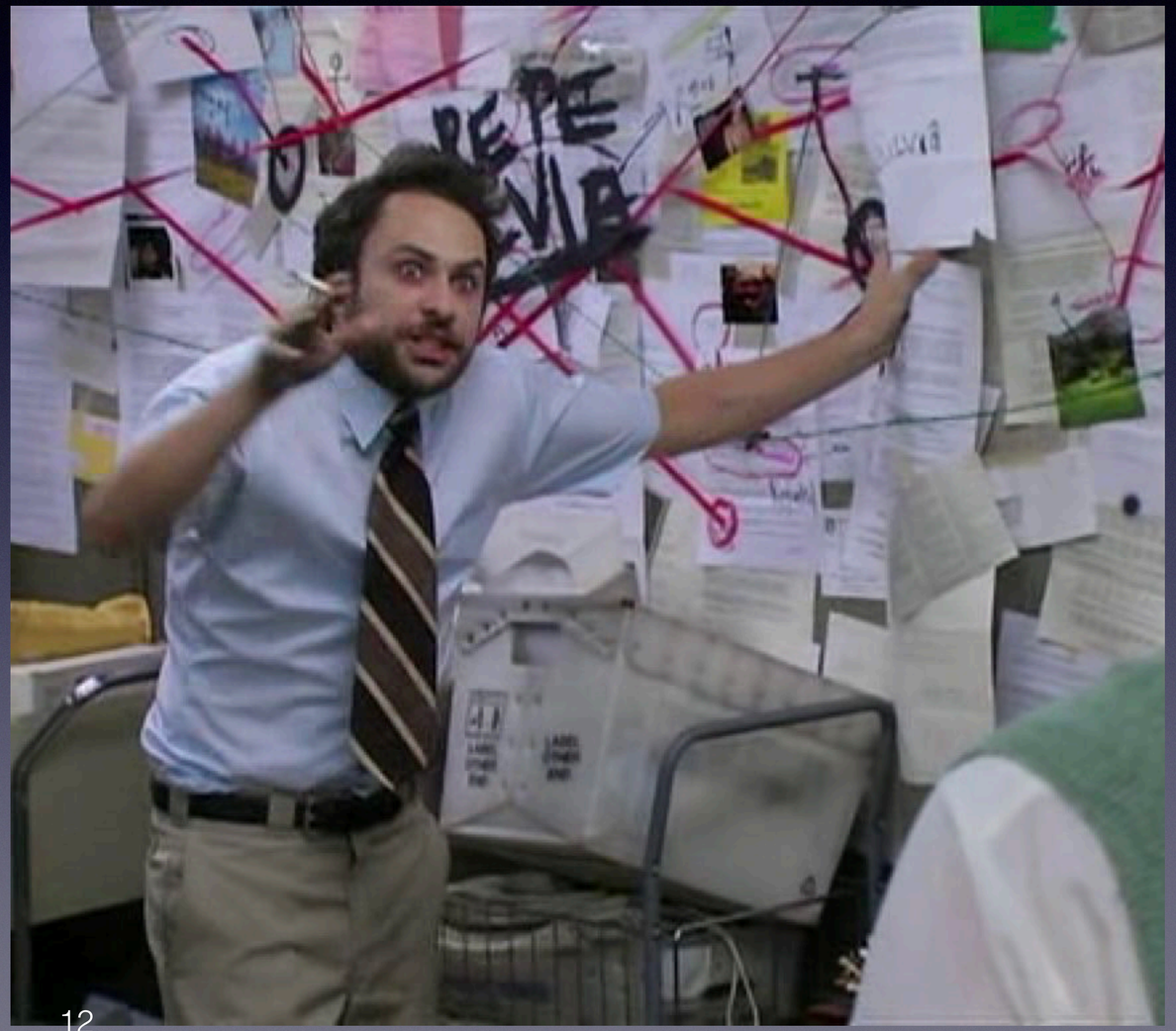
```
async function co_coutine() :Promise<number> {
    let res :number  = await timersPromises.setTimeout( handler: 2000, timeout: 'resul1');
    console.log("Co Routine 1", res);

    res = await timersPromises.setTimeout( handler: 2000, timeout: 'result2');
    console.log("Co Routine 2", res);

    res = await timersPromises.setTimeout( handler: 2000, timeout: 'result3');
    console.log("Co Routine 3", res);
    return res;
}
```

So lets use C++20 Coroutines





Co-Routines TL;DR

- C++20 Standard [dcl.fct.def.coroutine]: „A function is a coroutine if its function-body encloses a coroutine-return-statement (8.7.4), an await-expression (7.6.2.3), or a yield-expression (7.6.17).“ (Draft N4861 for ISO/IEC 14882:2020)
- You want to use
  - boost::asio::co_spawn to call a coroutine, usually with boost::asio::detached and a boost::asio::strand<boost::asio::io_context::executor_type> as executor
  - A return value that fullfills the C++20 Coroutine Concept like boost::asio::awaitable

From the old

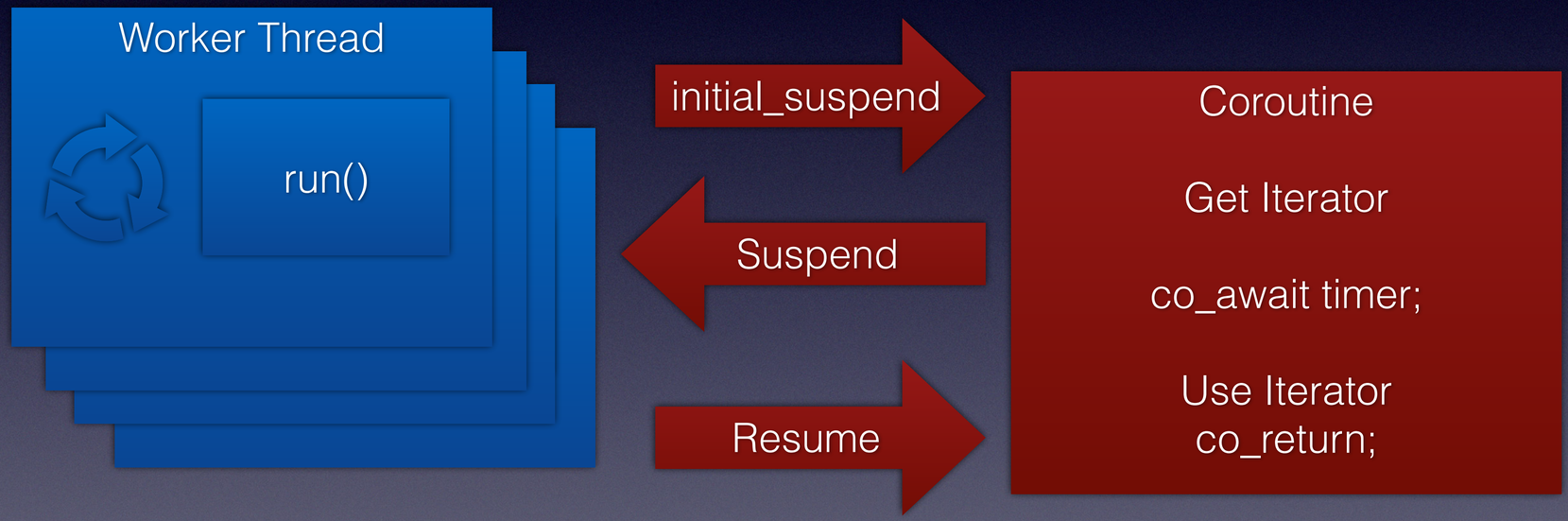
```
std::future<int> scope_stuff::old_async() {
    sync_ops example_instance; auto value = 7;
    this->timer_.expires_after( expiry_time: boost::asio::chrono::milliseconds( rep: 5));
    this->timer_.async_wait( token: [self : shared_ptr<scope_stuff>  = shared_from_this(), example_instance, value]
    (const auto &boost_error : const error_code & ) mutable -> void  {
        auto value2 : int  = example_instance.op2( input: value);
        self->timer_.expires_after( expiry_time: boost::asio::chrono::milliseconds( rep: 5));
        self->timer_.async_wait( token: [self, example_instance, value2]
                                   (const auto &boost_error) mutable{
            auto value3 : int  = example_instance.op3( input: value2);
            self->promise.set_value( r: value3);
        });
    });
    return this->promise.get_future();
}
```

To the new

```
boost::asio::awaitable<int> scope_stuff::getValue() {
    sync_ops example_instance; auto value = 7;
    this->timer_.expires_after( expiry_time: boost::asio::chrono::milliseconds( rep: 5));
    co_await this->timer_.async_wait( token: boost::asio::use_awaitable);
    value = example_instance.op2( input: value);
    this->timer_.expires_after( expiry_time: boost::asio::chrono::milliseconds( rep: 5));
    co_await this->timer_.async_wait( token: boost::asio::use_awaitable);
    value = example_instance.op3( input: value);

    co_return value;
}
```

Important


Worker Thread
run()
initial_suspend
Suspend
Resume
Coroutine
Get Iterator
co_await timer;
Use Iterator
co_return;

Demo: Dragons beware

- https://github.com/Superlokkus/coroutine
-

What we learned

- Its best to always include a co_return
- The calll operator does not behave as you expect in all cases anymore
- Execution is suspended during a await, a strand is released meanwhile!
- Magic happens in the return value
- **(Edit after Talk): Use a custom completion handler for co_spawn to handle/rethrow unmatched exceptions leaking out of the top coroutines (see in code example)**

Async keyword

- C++ misses the async keyword for function declarations, it's signaled by the keyword in the body

```
Hello World!
```

```python
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

```javascript
async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
  // Expected output: "resolved"
}

asyncCall();
```

What helped me

- https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html
- https://think-async.com/Asio/asio-1.22.1/doc/asio/overview/composition/coro.html
- https://www.boost.org/doc/libs/1_82_0/doc/html/boost_asio/overview/composition/cpp20_coroutines.html

Thank you

- https://stackoverflow.com/q/76950493/3537677
- https://stackoverflow.com/q/77060868/3537677
- https://stackoverflow.com/q/76922347/3537677