A Pattern Language for Expressing Concurrency

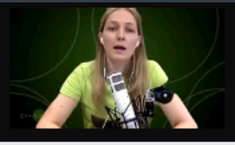
LUCIAN RADU TEODORESCU

A Pattern Language for Expressing Concurrency

lucteo.ro/pres/2022-cppcon/

LUCIAN RADU TEODORESCU
GARMIN

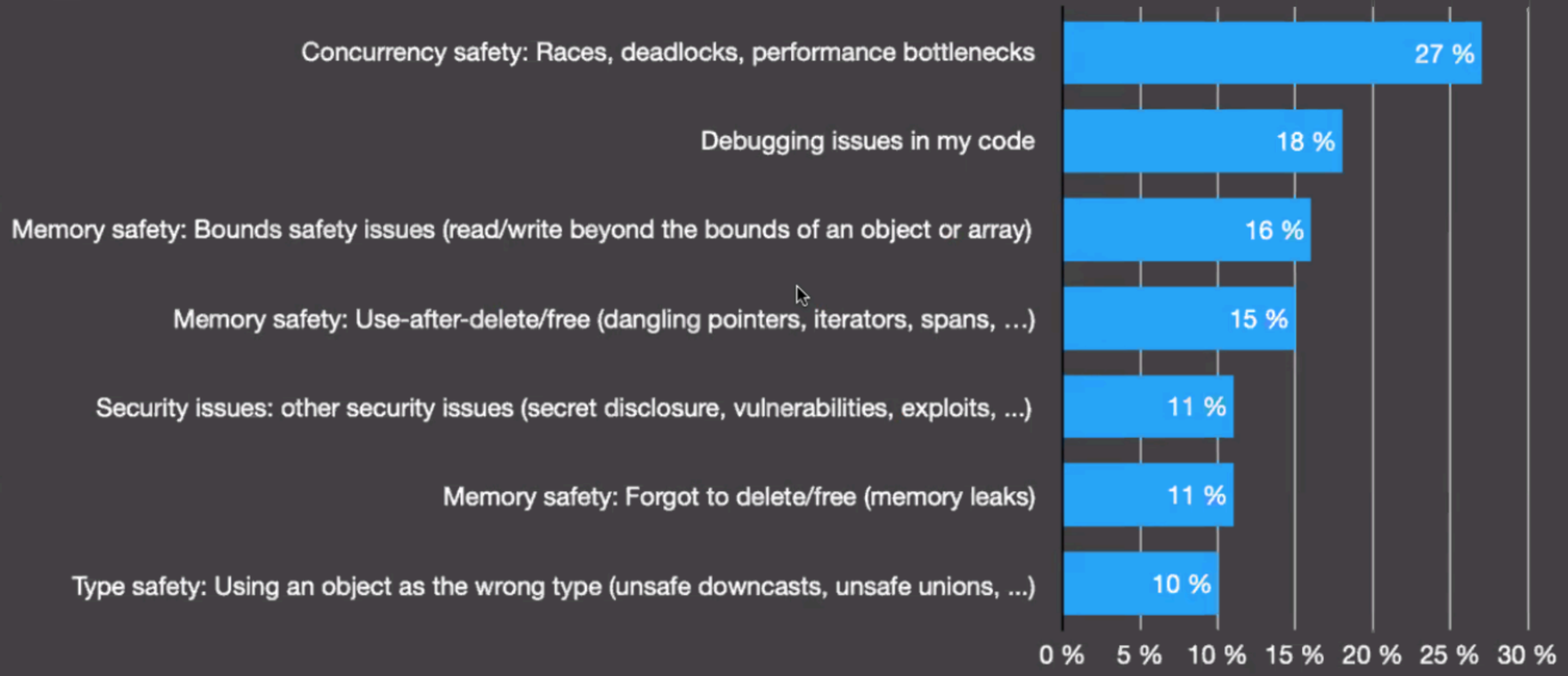
concurrency is **HARD**



Code analysis++ EN - Google

docs.google.com/presentation/d/1rfdksmy28o0Oc4EA1MnP2dQE-vyf7G4MDbZKPj0AAmU/edit#slide=id.p

C++ development frustration: safety & security



we want

safety
performance
structured approach

P2300 — `std::execution` (senders and receivers)

C++ proposal

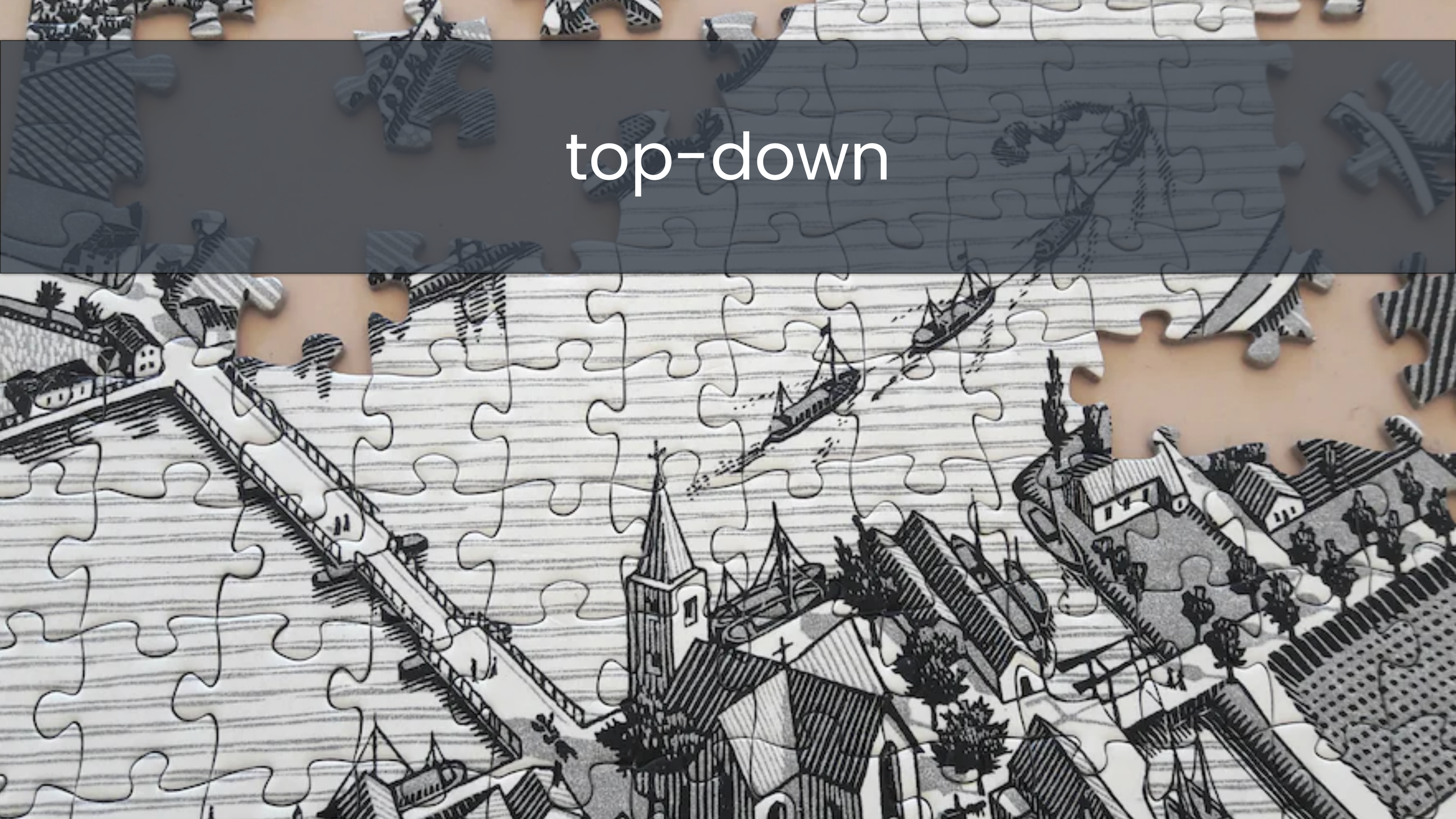
did not make it to C++23

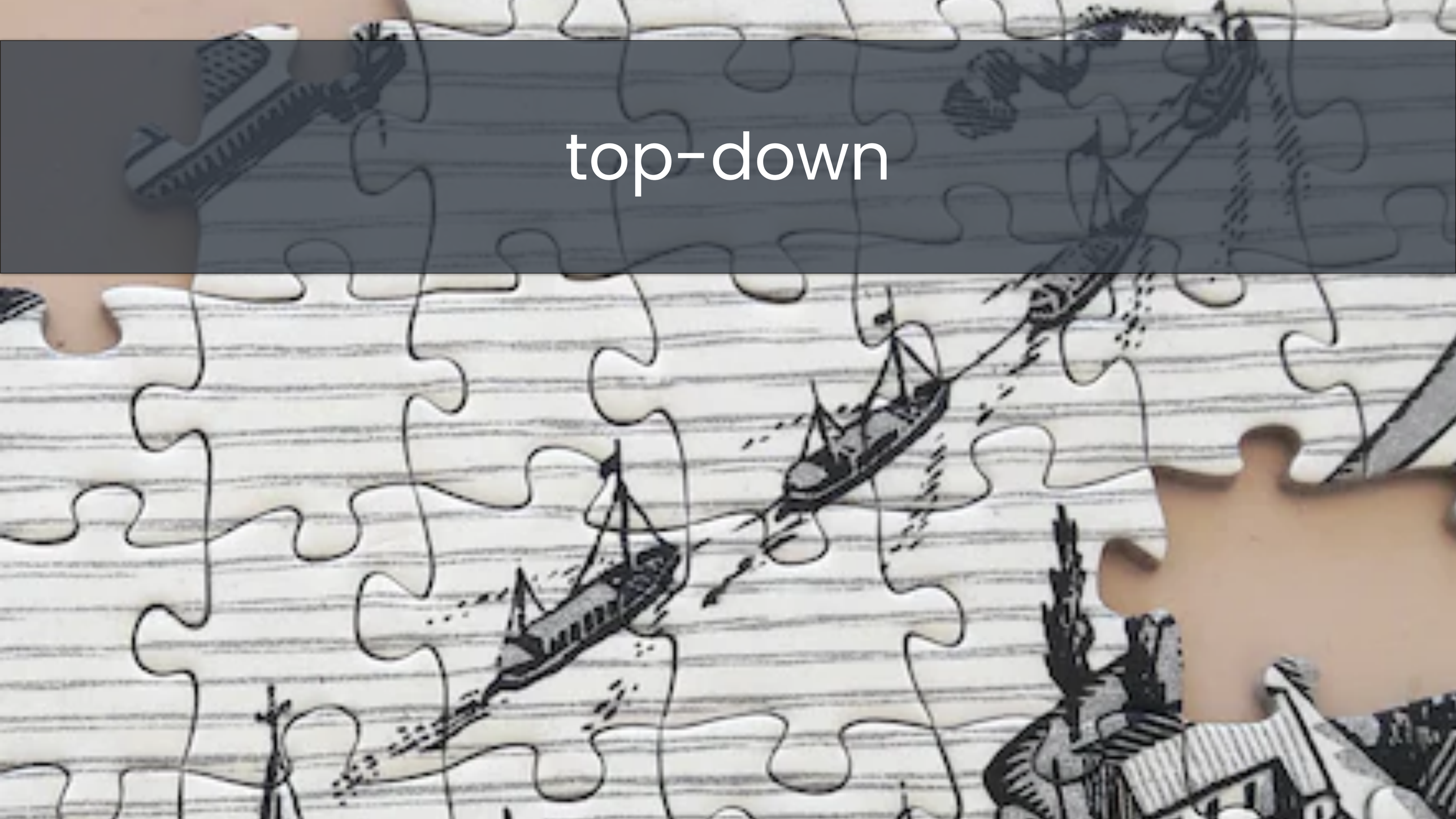
approach

abstractions

shapes

solving the puzzle


top-down


top-down

Agenda

Structured Concurrency

Introductory Examples

Starting with Patterns

Extending the Patterns Repertoire

Composition with Patterns

Structured Concurrency

An Introduction

Structured **Concurrency**

extending the ideas from **Structured Programming**

what is **Structured Programming**?

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness

functions

no GOTOs

sequence, selection, repetition

APIC Studies in Data Processing No. 8

STRUCTURED PROGRAMMING

O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare

Academic Press
London New York San Francisco
A Subsidiary of Harcourt Brace Jovanovich, Publishers

1972

concurrency with **threads**

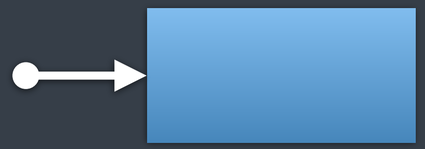
primitives: threads & mutexes

concurrency with threads: **structured**?

~~abstractions as building blocks~~	no
~~recursive decomposition~~	no
~~local reasoning~~	no
~~single entry, single exit point~~	-
~~soundness~~ and completeness	½

concurrency with **raw tasks**

primitives: tasks
(independent units of work)



raw tasks — **structured**?

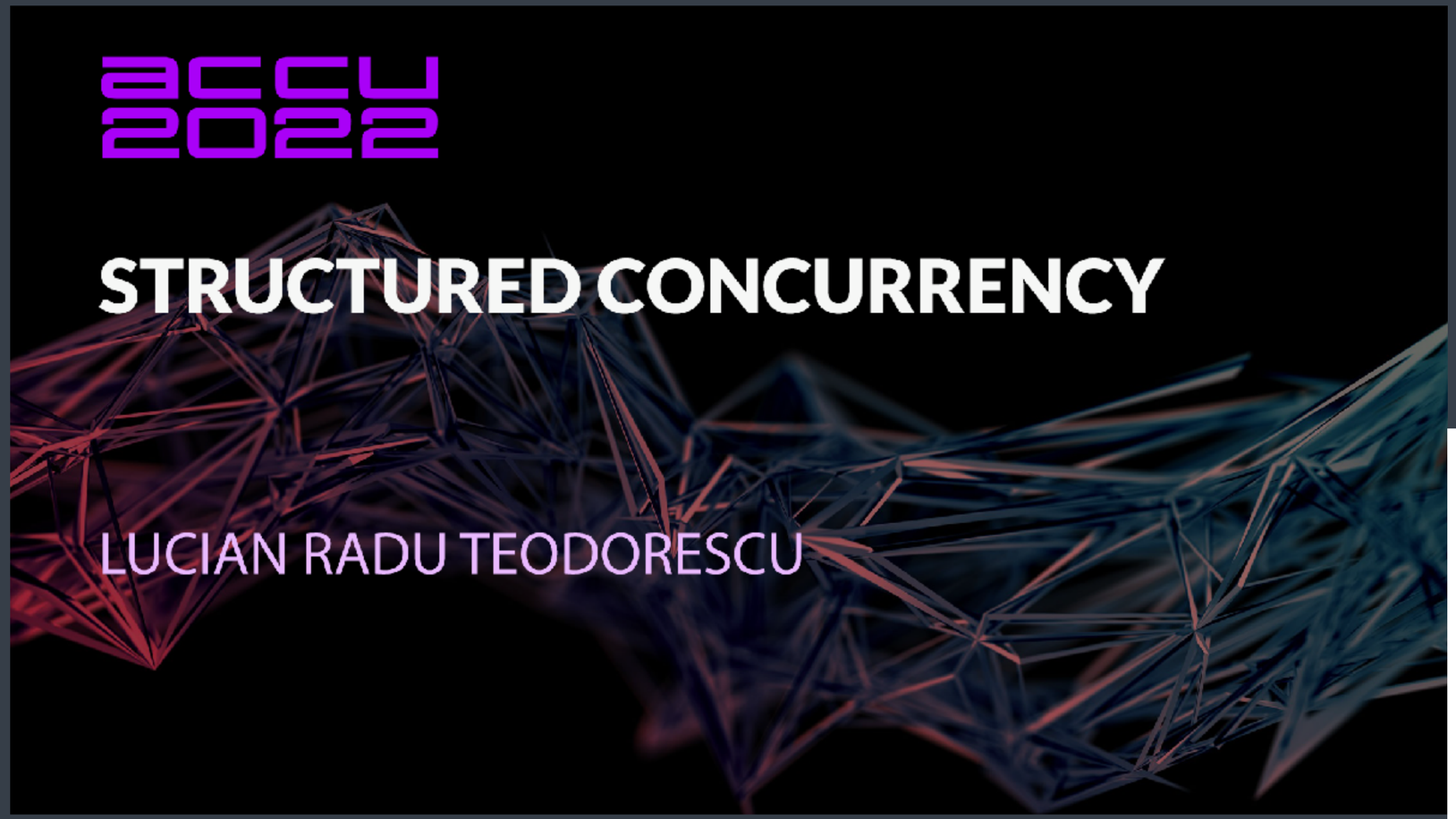
abstractions as building blocks	yes
~~recursive decomposition~~	no
local reasoning	yes
~~single entry, single exit point~~	no
soundness and completeness	yes

P2300 — `std::execution` (senders and receivers)

primitives: senders

P2300 — **Structured Concurrency**

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness



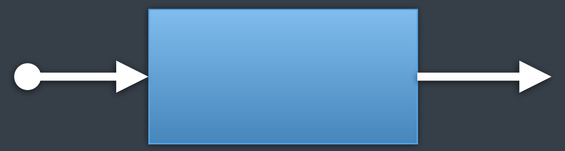
https://www.youtube.com/watch?v=Xq2IMOPjPs0

senders describe **computations**

any chunk of work,
with one entry and one exit point

computations

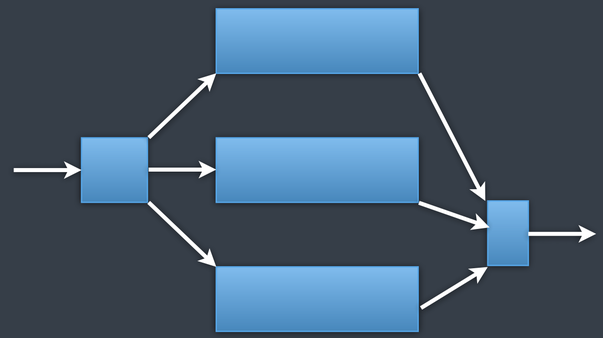
a task



computations

a task

tasks over multiple threads



computations

a task

tasks over multiple threads

group of computations

computations

a task

tasks over multiple threads

group of computations

the entire application

computations

a task
tasks over multiple threads
group of computations
the entire application

any chunk of work,
with one entry and one exit point

functions

same thread

computations

entry thread ≠ exit thread

computations

generalisation of functions

computations are for **concurrency**
what functions are for **Structured** Programming

senders model computations

what about **coroutines**?

coroutines — **Structured Concurrency**

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness

coroutine tasks

≡

senders

Introductory Examples

2

Hello, concurrent world!

```
namespace ex = std::execution;

auto say_hello() {
    return ex::just() // just a signal
           | ex::then([] {
                std::printf("Hello, concurrent world!\n");
                return 0;
             });
}

int main() {
    auto [r] = std::this_thread::sync_wait(say_hello()).value();
    return r;
}
```

shapes of the functions

`just`	() → *sender*
`then`	(*sender*, *ftor*) → *sender*
`sync_wait`	(*sender*) → `optional<tuple<`*vals*`...>>`

sender

describes work
eventually produces a result

(similar to a future)

Hello, concurrent world!

```
namespace ex = std::execution;

auto say_hello() {
    return ex::just() // just a signal
           | ex::then([] {
                std::printf("Hello, concurrent world!\n");
                return 0;
             });
}

int main() {
    auto [r] = std::this_thread::sync_wait(say_hello()).value();
    return r;
}
```

Hello, concurrent world! (2)

```
task<int> say_hello() {
    std::printf("Hello, concurrent world!\n");
    co_return 0;
}

int main() {
    auto [r] = std::this_thread::sync_wait(say_hello()).value();
    return r;
}
```

coroutine tasks

≡

senders

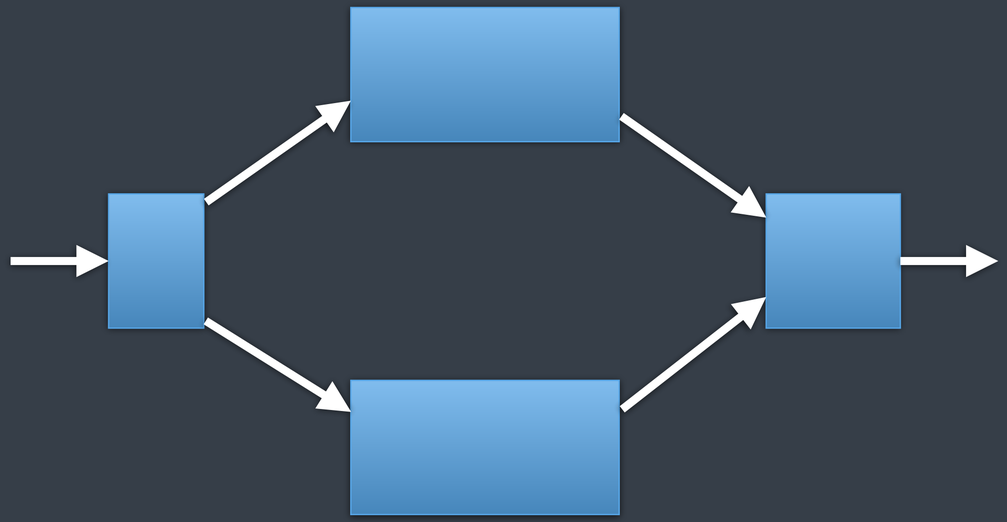
rock, paper, scissors

2 players doing work in parallel

rock, paper, scissors

```
enum class shape { rock, paper, scissors };

shape player_choose() { return static_cast<shape>(rand() % 3); }

void print_result(shape r1, shape r2) { … }

void play_rock_paper_scissors() {
    static_thread_pool pool{8};
    ex::scheduler auto sched = pool.get_scheduler();

    ex::sender auto game_work = ex::when_all(                //
            ex::schedule(sched) | ex::then(player_choose), //
            ex::schedule(sched) | ex::then(player_choose)  //
    );
    auto [r1, r2] = sync_wait(std::move(game_work)).value();
    print_result(r1, r2);
}
```

rock, paper, scissors



concurrency analysis

threads are hidden
expressed concurrency as a graph
no explicit synchronization
framework ensures efficiency

how to **describe**

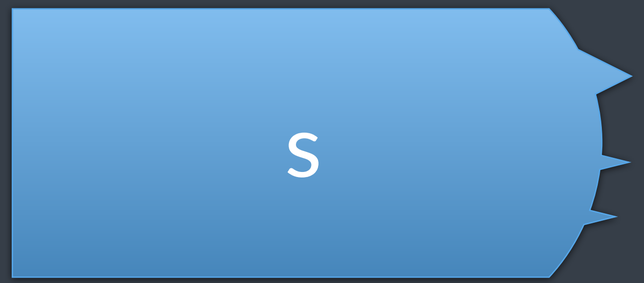
concurrency **?**

Starting with Patterns

3

functions

senders



exit point for **functions**

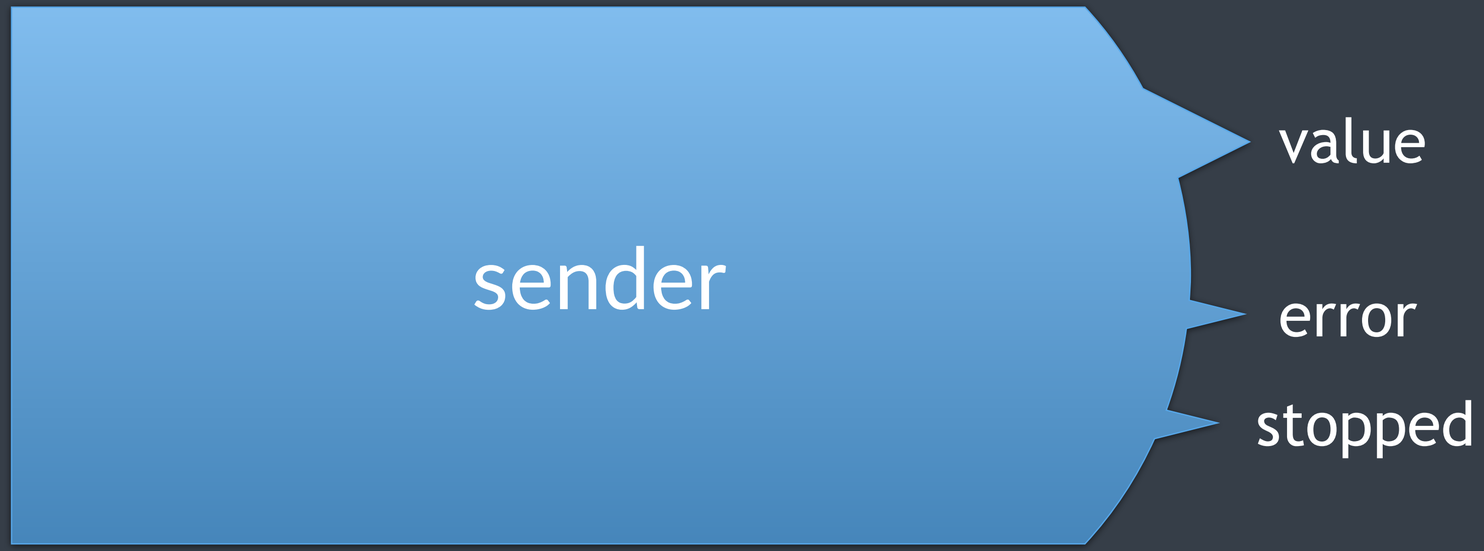
return value (or void)

exceptions

stopped — i.e., `exit()` called

exit point for **senders**

```
set_value(vals…)
set_error(err)
set_stopped()
```


sender
value
error
stopped

P1: create value

purpose:

- transform a value into a sender

when to use:

- inject a value into a sender flow
- start a sender flow

creating value

```
void create_value_example() {
    ex::sender auto s = ex::just(13);

    auto [r] = sync_wait(s).value();
    assert(r == 13);
    std::printf("%d\n", r);
}
```

multiple values

```
void create_value_example2() {
    ex::sender auto s = ex::just(17, 19);

    auto [a, b] = sync_wait(s).value();
    assert(a == 17);
    assert(b == 19);
    std::printf("%d, %d\n", a, b);
}
```

no value, just a signal

```
void create_value_example3() {
    ex::sender auto s = ex::just();

    sync_wait(s); // finishes immediately
}
```


just(val…)
val…

variations

```
just_error(e)
just_stopped()
```

P2: synchronous wait

purpose:

- wait for a sender to complete

when to use:

- at the location when the computation needs to be complete; i.e., `main()`

when NOT to use:

- too many small function

waiting for a value

```
void create_value_example() {
    ex::sender auto s = ex::just(13);

    auto [r] = sync_wait(s).value();
    assert(r == 13);
    std::printf("%d\n", r);
}
```


sync_wait()


just(val…)
sync_wait()

P3: transforming values

purpose:

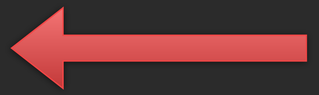
- apply a function to transform a value

when to use:

- when needing to perform actions on a sender
- when values need to be transformed

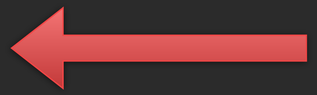
transforming values

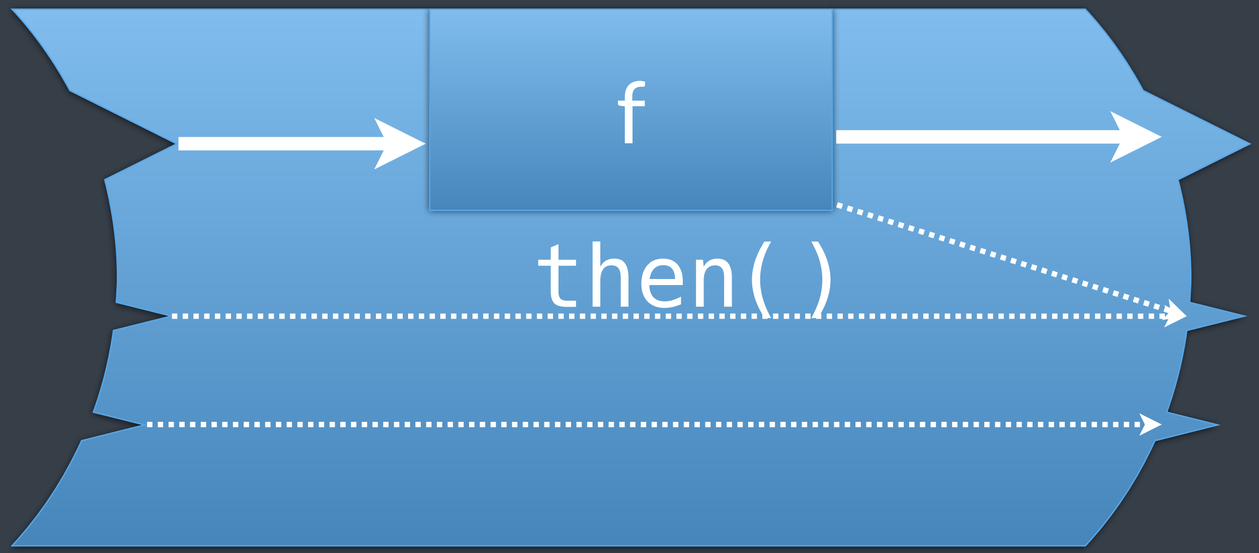
```
int process_value(int x) {
    std::printf("I've got value: %d\n", x);
    return x*x;
}
void then_example() {
    ex::sender auto s = ex::just(13);
    ex::sender auto s2 = ex::then(s, process_value);

    auto [r] = sync_wait(std::move(s2)).value();
    assert(r == 169);
    std::printf("%d\n", r);
}
```



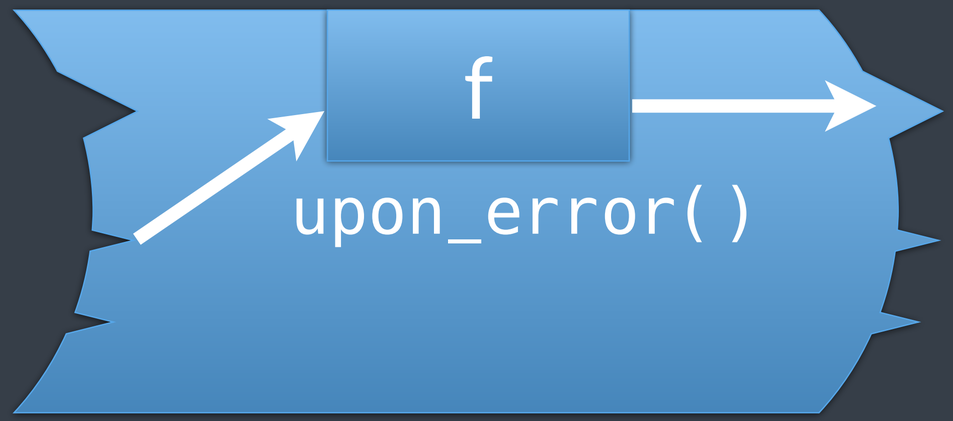
transforming values

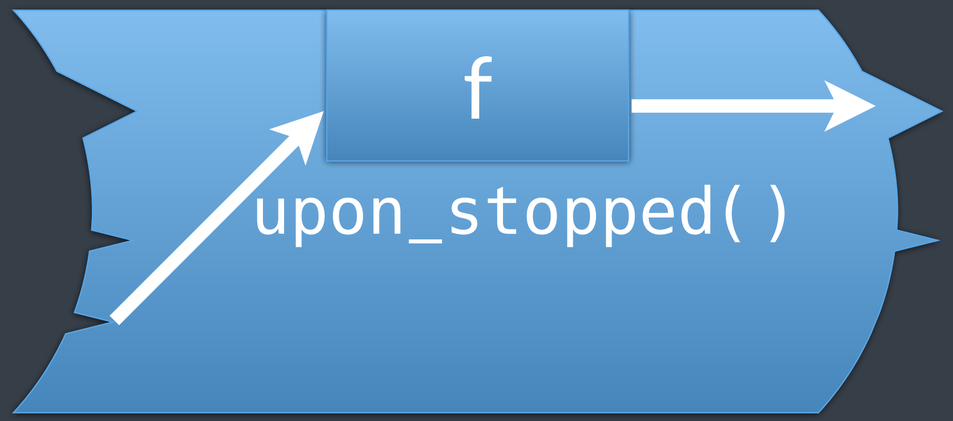
```
int process_value(int x) {
    std::printf("I've got value: %d\n", x);
    return x*x;
}
void then_example2() {
    ex::sender auto s = ex::just(13)
                      | ex::then(process_value);

    auto [r] = sync_wait(std::move(s)).value();
    assert(r == 169);
    std::printf("%d\n", r);
}
```




f
then()

variations





P4: joining

purpose:

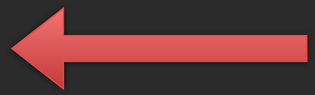
- join multiple senders into a single one

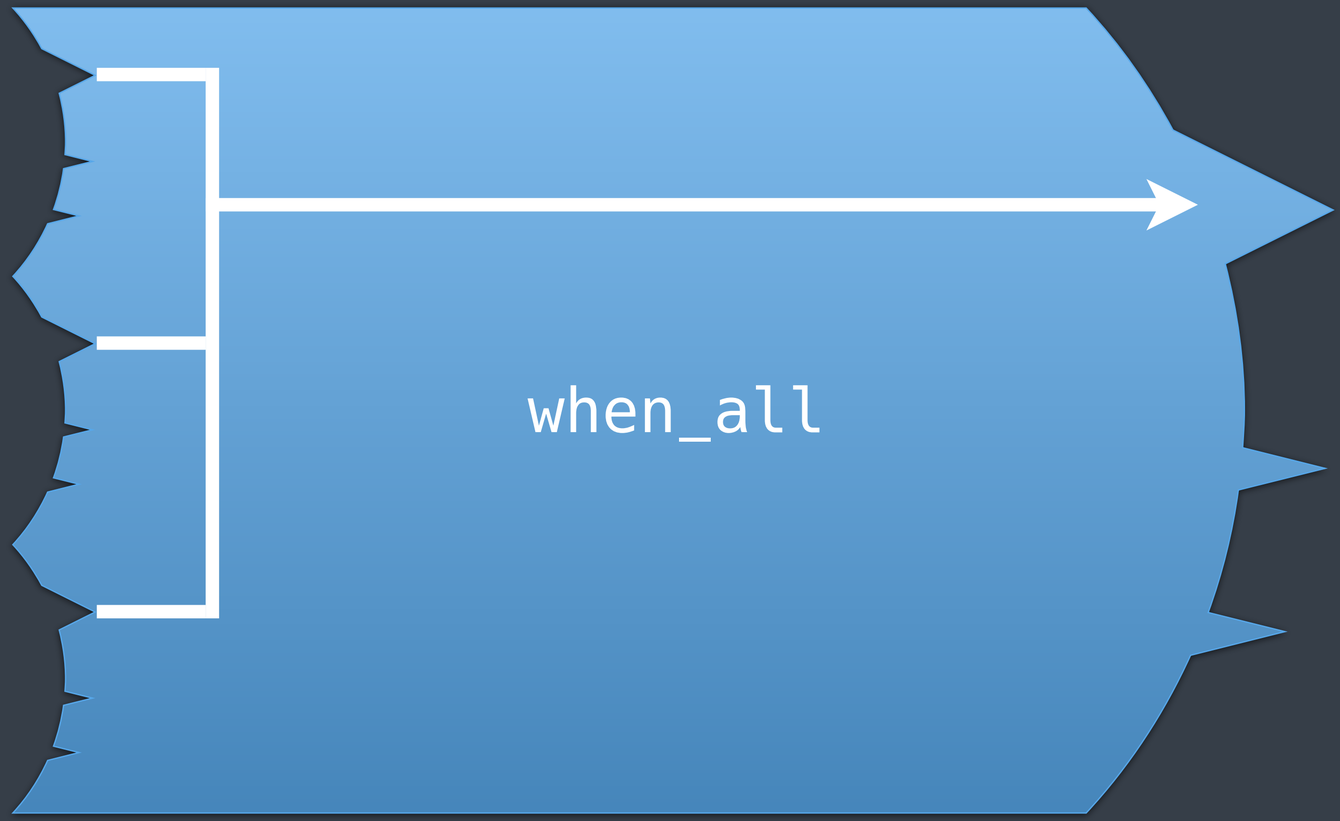
when to use:

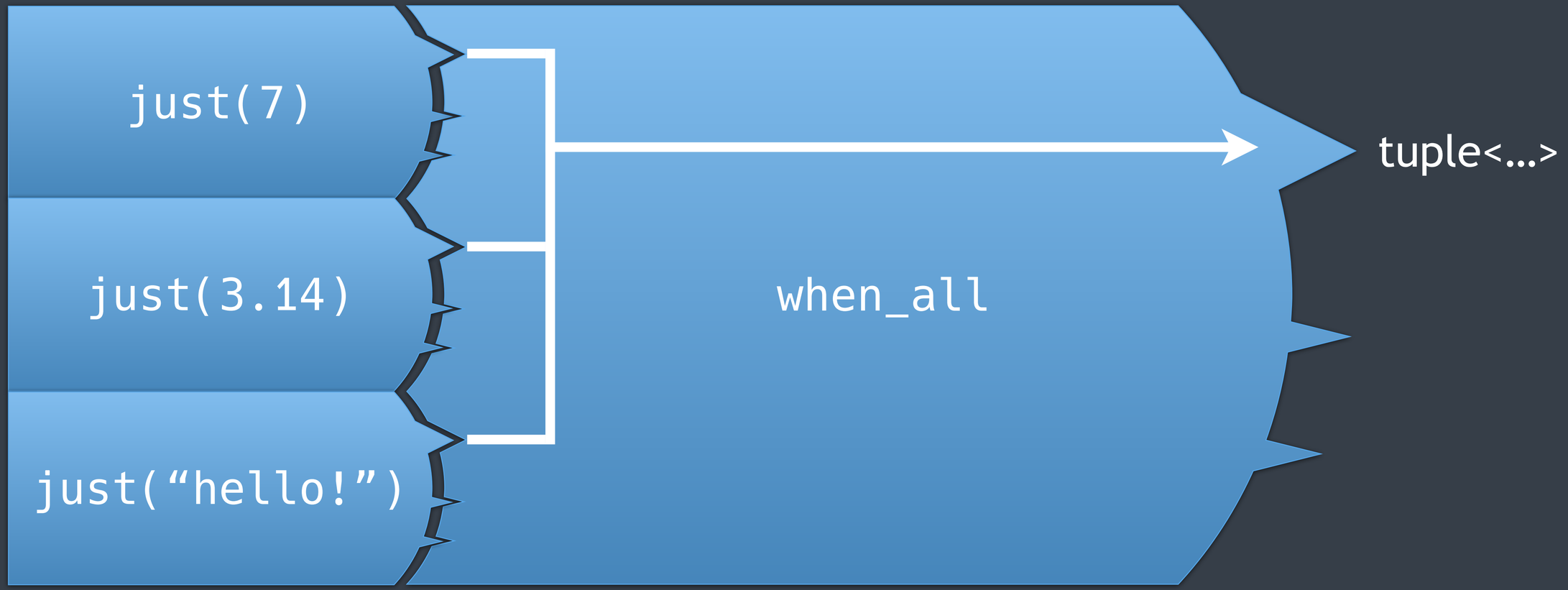
- combine parallel work
- detect the finish of multiple execution paths

joining senders

```
void join_example() {
    ex::sender auto s = ex ::when_all( //
            ex::just(7),               //
            ex::just(3.14),            //
            ex::just("hello!")         //
    );

    auto [i, d, str] = sync_wait(std::move(s)).value();
    std::printf("%d, %g, %s\n", i, d, str);
}
```




when_all


just(7)
just(3.14)
just(“hello!”)
when_all
tuple<...>

P5: scheduling

purpose:

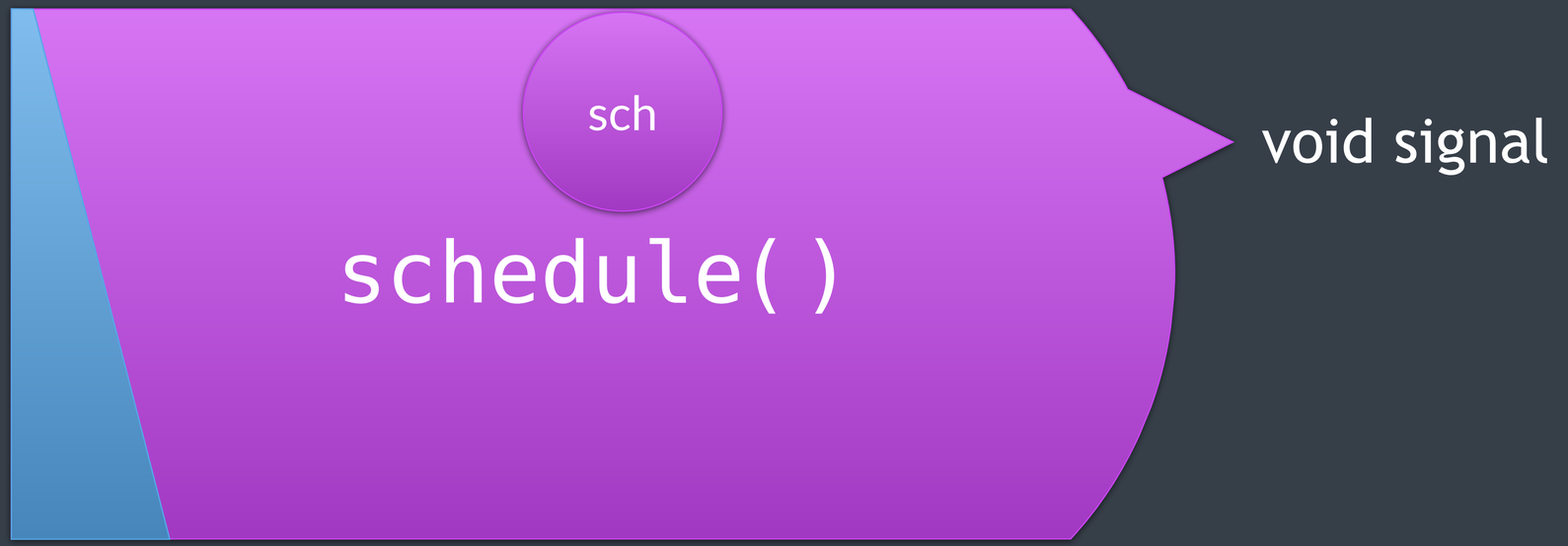
- start work on a different execution context

when to use:

- start parallel work chain
- transform a scheduler into a sender

scheduling work

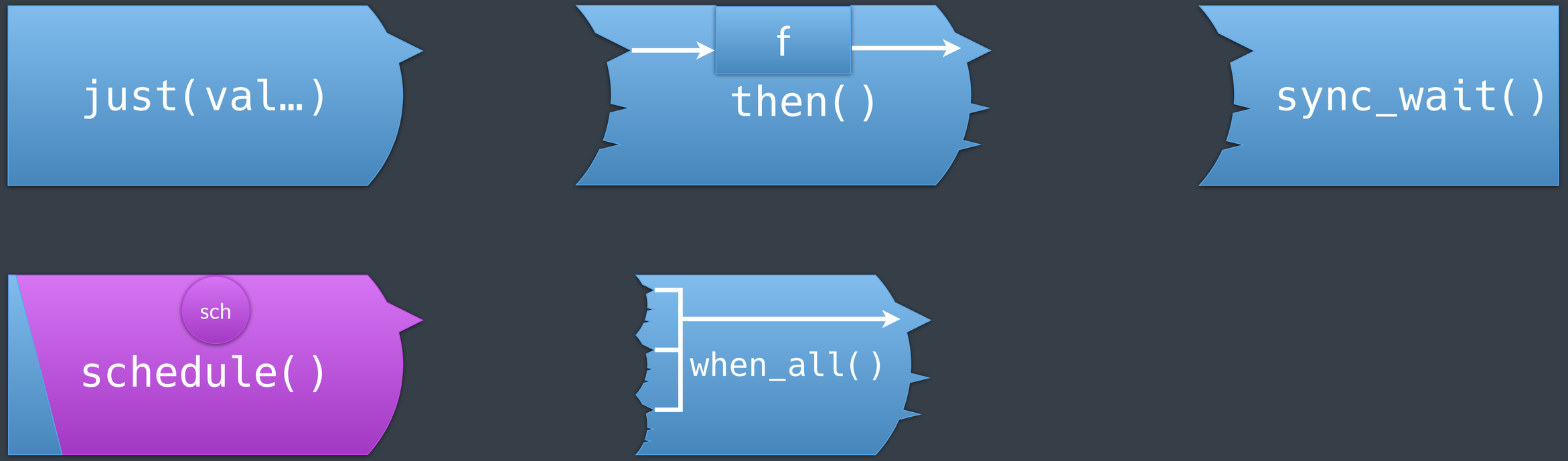
```
void schedule_example() {
    static_thread_pool pool{8};
    ex::scheduler auto sch = pool.get_scheduler();

    ex::sender auto s = ex::schedule(sch)
                      | ex::then([]{std::printf("Hello from another thread!");});

    sync_wait(std::move(s));
}
```


sch
schedule()
void signal


sch
schedule()
f
then()
sync_wait()

variation

patterns so far



Extending the Patterns Repertoire

4

P6: composing senders

purpose:

- allow composition of senders

when to use:

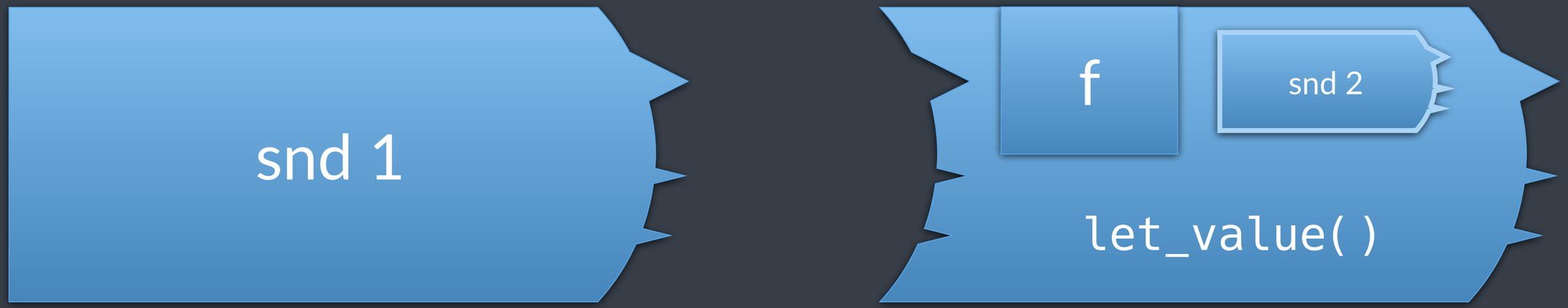
- compose senders (monadic bind usecases)
- ensure data is alive for the entire lifetime of a sender

the problem of composition

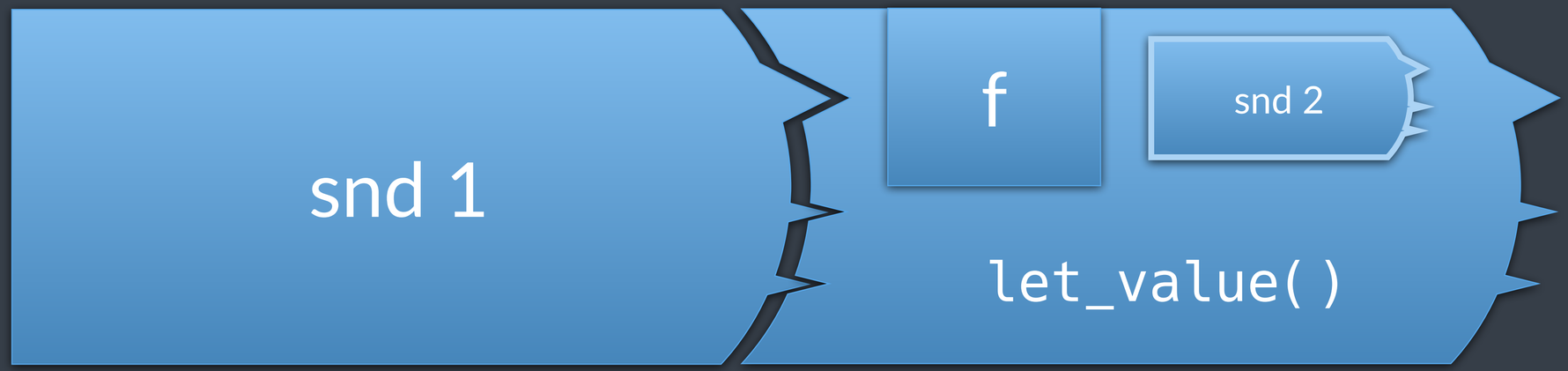

f
let_value

how composition works

how composition works



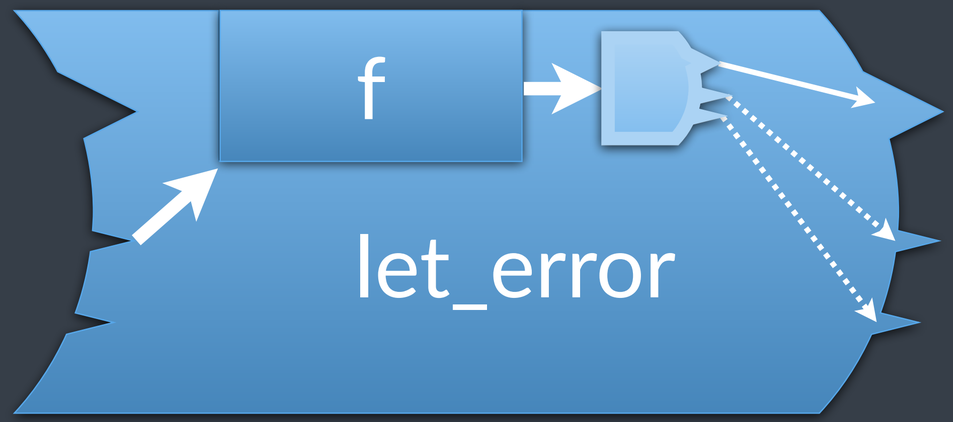
how composition works



composing senders

```
ex::sender auto schedule_request_start(read_requests_ctx ctx) { … }
ex::sender auto validate_request(const http_request& req) { … }
ex::sender auto handle_request(const http_request& req) { … }
ex::sender auto send_response(const http_response& resp) { … }

ex::sender auto request_pipeline(read_requests_ctx ctx) {
    return
        schedule_request_start(ctx)
        | ex::let_value(validate_request)
        | ex::let_value(handle_request)
        | ex::let_value(send_response)
        ;
}
```

variants



graphical alternative

P7: starting senders in other contexts

purpose:

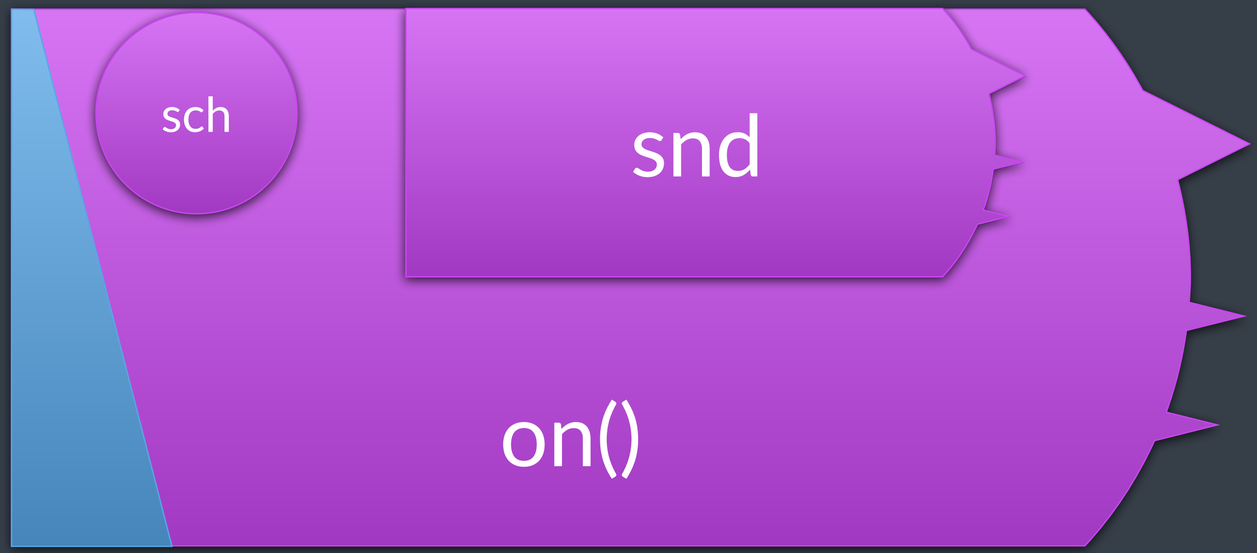
- start a sender on a different execution context

when to use:

- work needs to start in a specified scheduler

start sender in a new context

```
ex::sender auto do_read_from_socket() { … }


io_context io_threads;
ex::scheduler auto sch = io_threads.get_scheduler();

ex::sender auto snd = ex::on(sch, do_read_from_socket());

sync_wait(std::move(snd));
```


sch
snd
on()

equivalence

```
on(sch, snd)
```

≡

```
ex::schedule(sch) | ex::let_value([]{ return snd; })
```

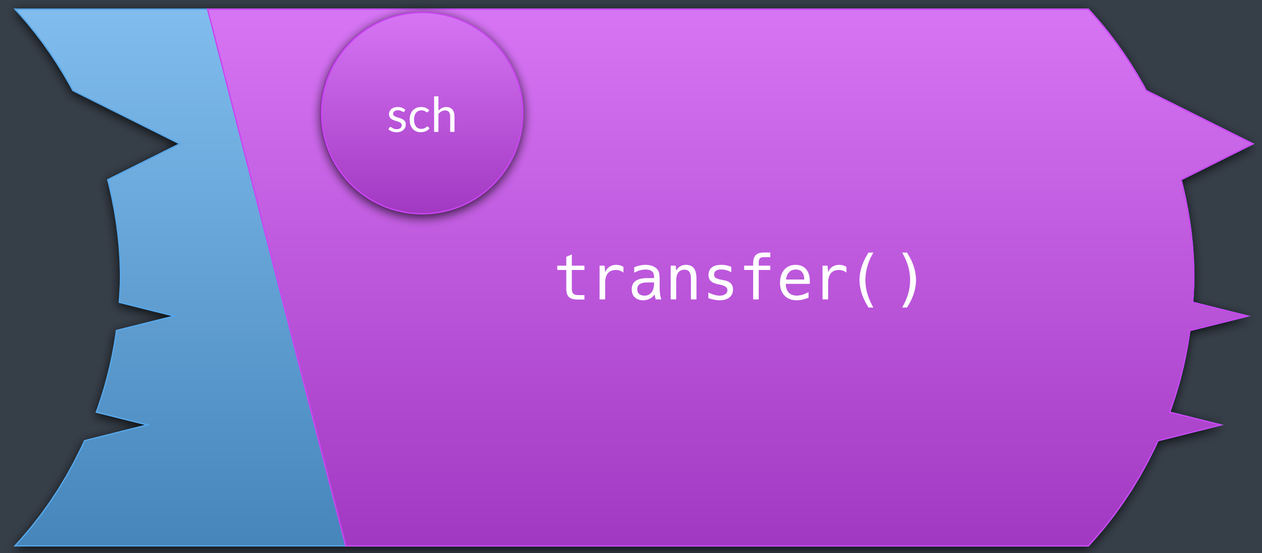
P8: transfer between contexts

purpose:

- transfer the work chain to a different execution context

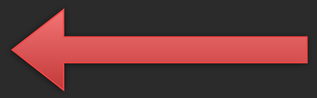
when to use:

- work needs to change execution context
- add new work on the same execution context


sch
transfer()

transfer between contexts

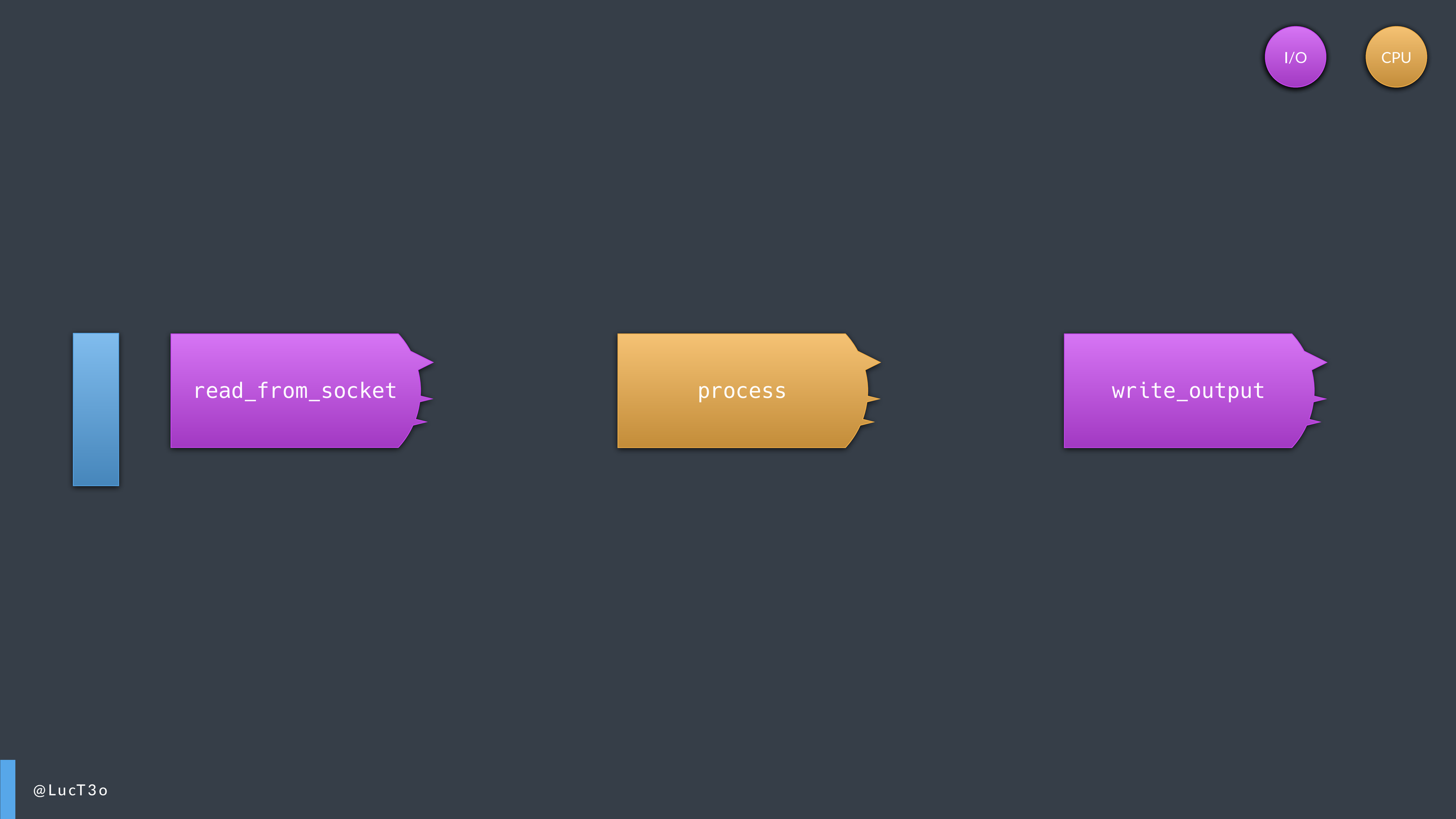
```
ex::sender auto read_from_socket() { … }
ex::sender auto process(in_data) { … }
ex::sender auto write_output(out_data) { … }

io_ontext io_threads;
static_thread_pool work_pool{8};
ex::scheduler auto sch_io = io_threads.get_scheduler();
ex::scheduler auto sch_cpu = work_pool.get_scheduler();

ex::sender auto snd
        = ex::on(sch_io, read_from_socket())
        | ex::transfer(sch_cpu)
        | ex::let_value(process)
        | ex::transfer(sch_io)
        | ex::let_value(write_output)
        ;

sync_wait(std::move(snd));
```

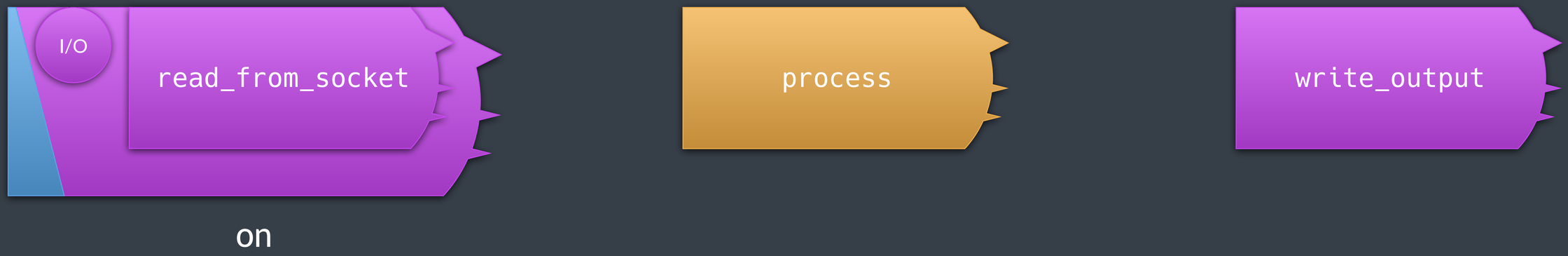



read_from_socket
process
write_output

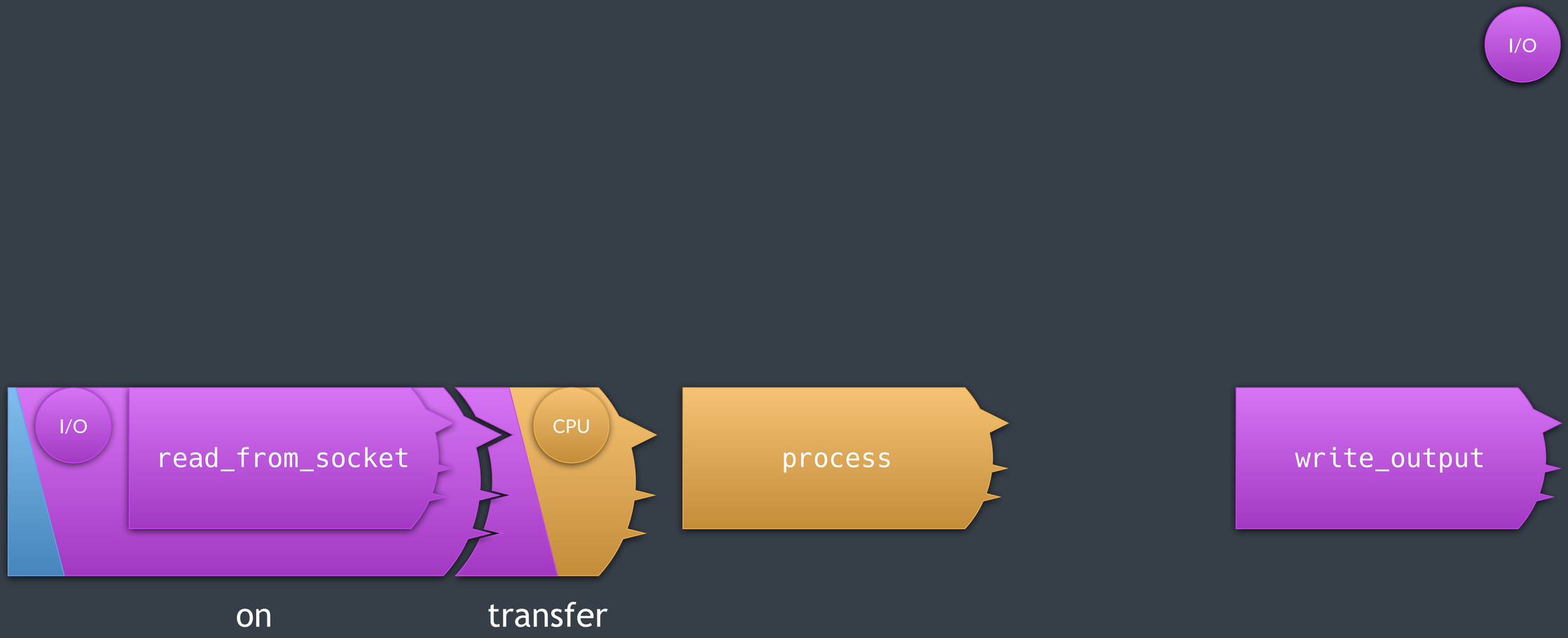

I/O
CPU

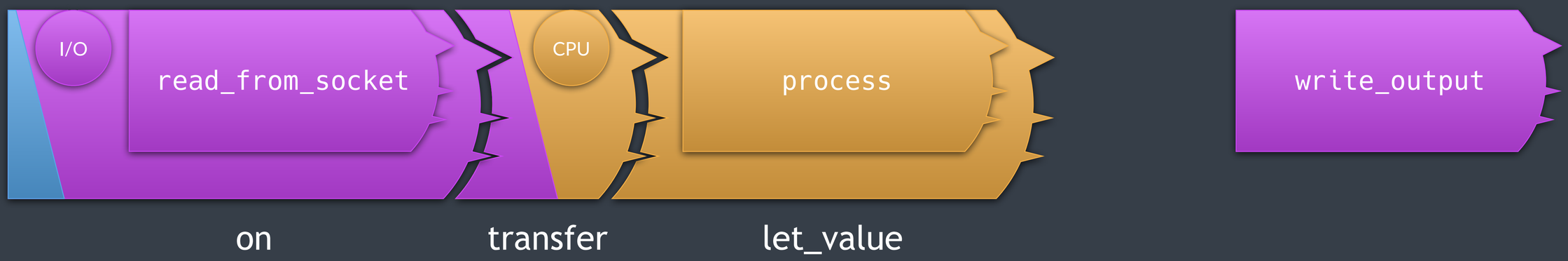

read_from_socket
process
write_output


I/O
CPU
read_from_socket
process
write_output

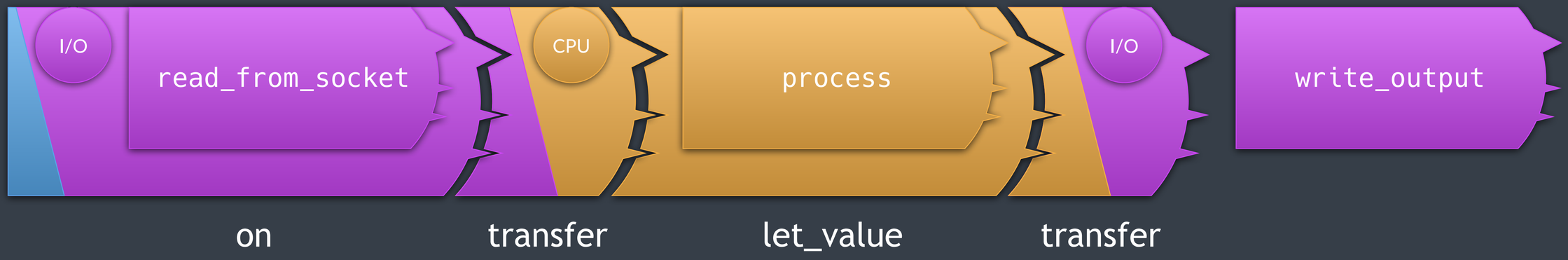


on

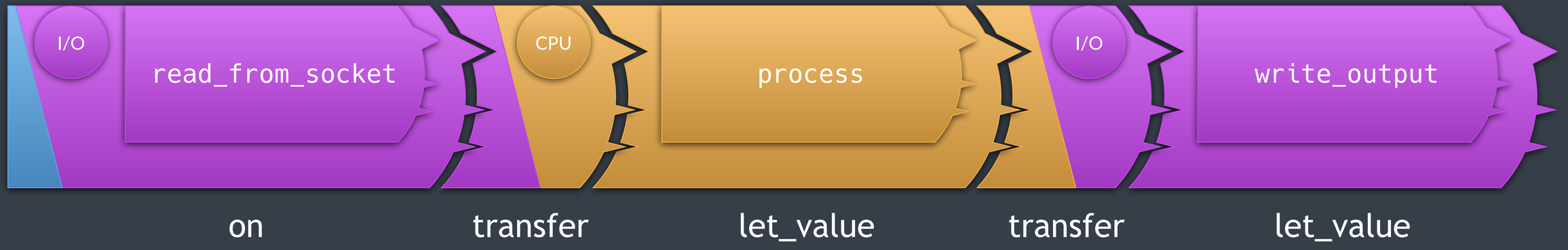

I/O
CPU
I/O
read_from_socket
CPU
process
write_output
on
transfer


I/O
CPU
I/O
read_from_socket
CPU
process
write_output
on
transfer
let_value

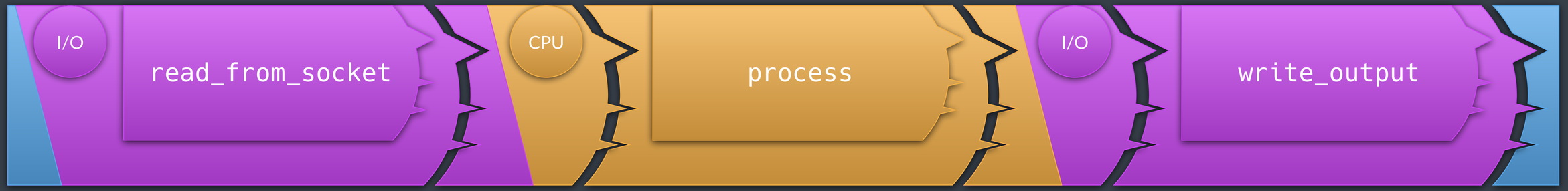
I/O CPU


I/O
read_from_socket
CPU
process
I/O
write_output
on
transfer
let_value
transfer

I/O CPU


I/O
read_from_socket
CPU
process
I/O
write_output
on
transfer
let_value
transfer
let_value

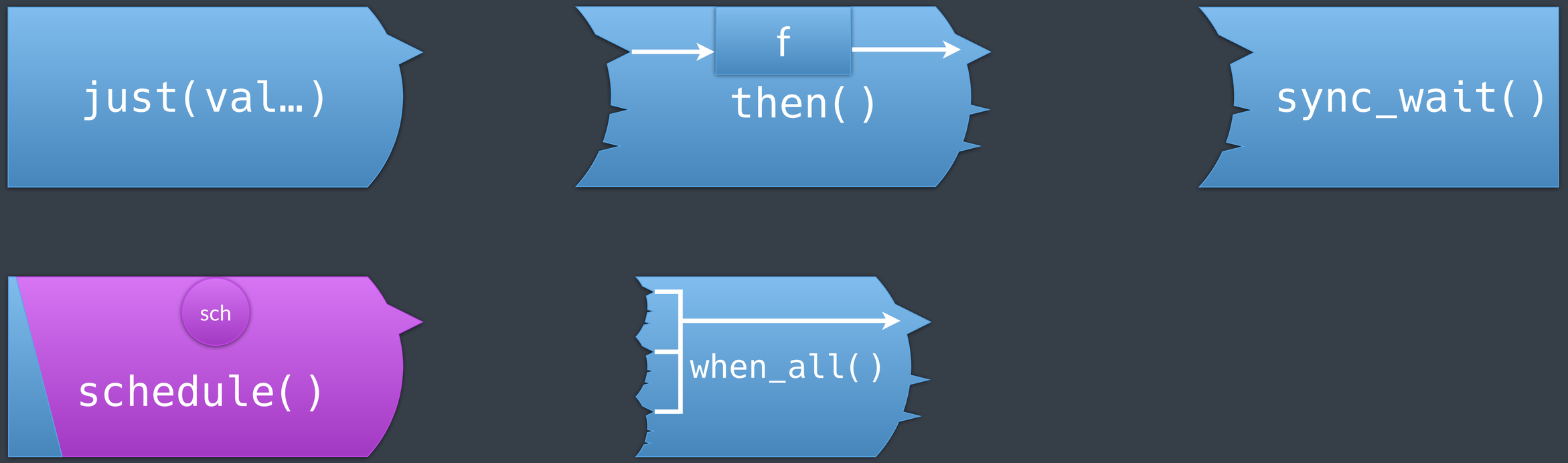
I/O CPU

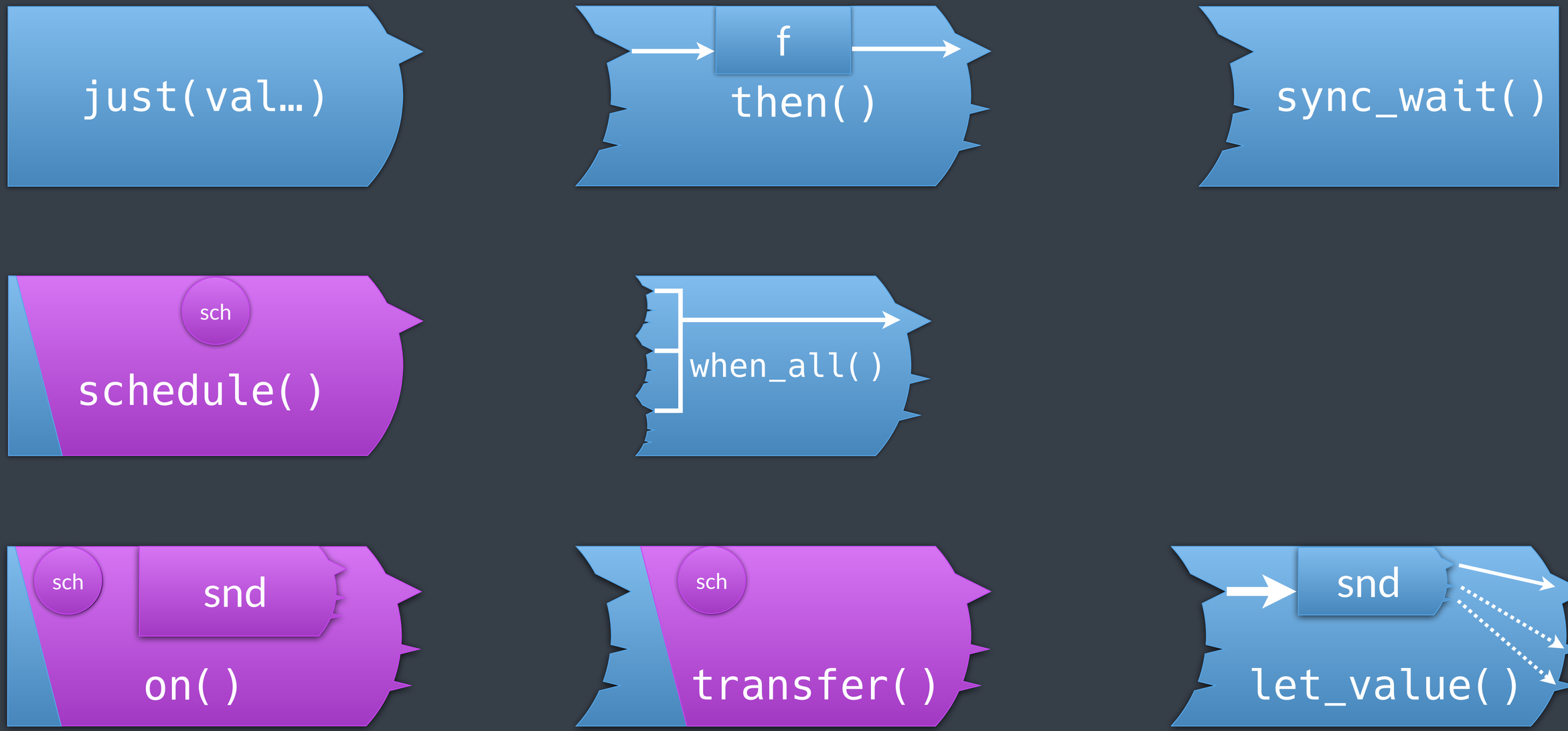


on transfer let_value transfer let_value sync_wait

more in P2300

- `bulk`
- `split, ensure_started`
- `start_detached`
- `into_variant, stopped_as_optional, stopped_as_error`
- `read`
- coroutine support


just(val…)
f
then()
sync_wait()
sch
schedule()
when_all()


just(val…)
f
then()
sync_wait()
sch
schedule()
when_all()
sch
snd
on()
sch
transfer()
snd
let_value()

Composition with Patterns

5

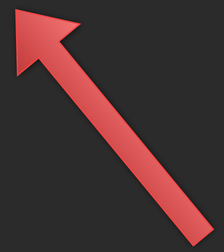
application

HTTP server
image processing

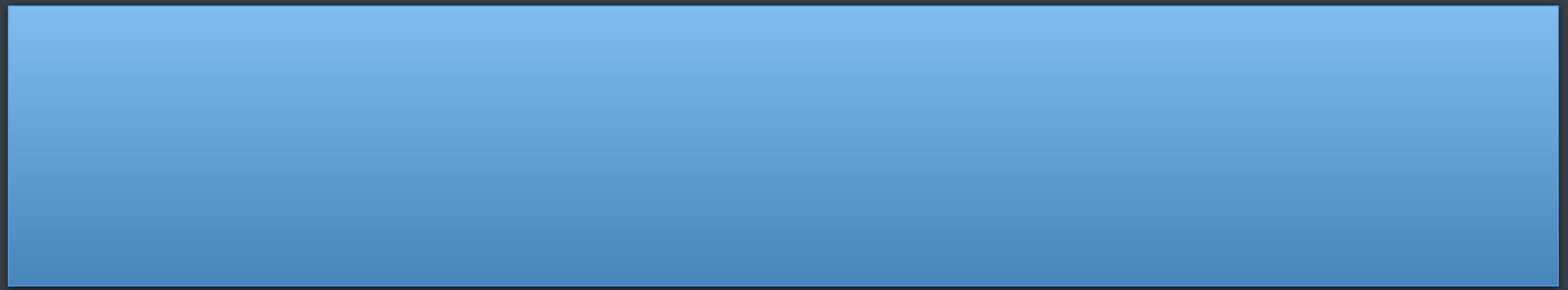
https://github.com/lucteo/structured_concurrency_example

the entire app is a sender

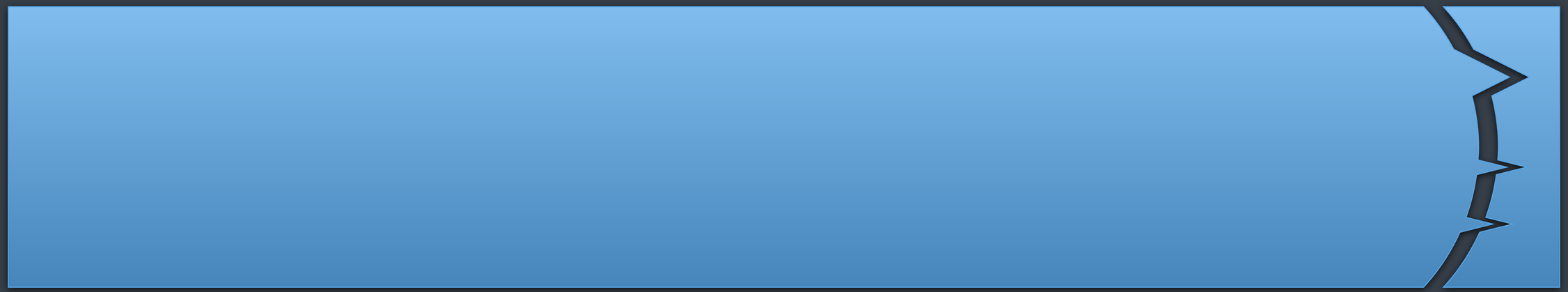
```
auto get_main_sender() {
    return ex::just() | ex::then([] {
        //...
        return 0;
    });
}

auto main() -> int {
    auto [r] = std::this_thread::sync_wait(get_main_sender()).value();
    return r;
}
```



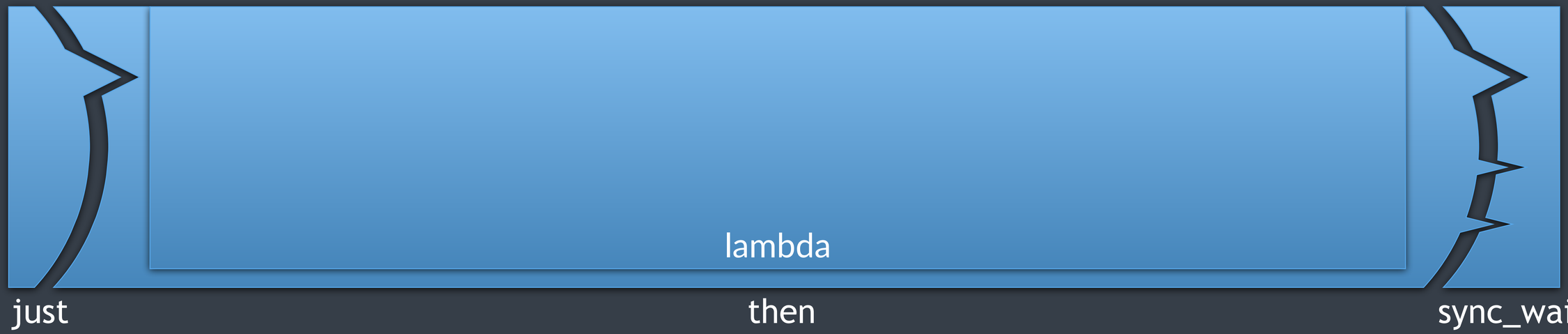
main()



main()



main()



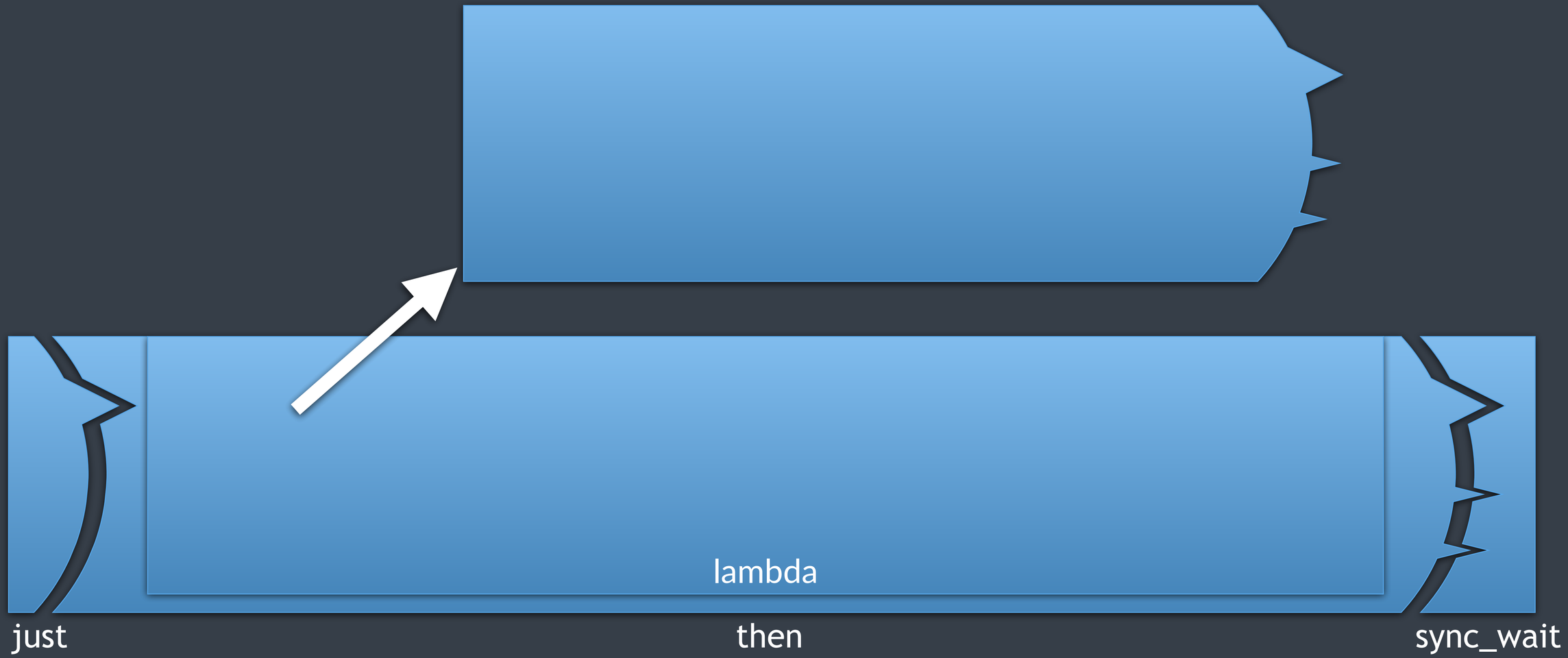
top-level logic

```
auto get_main_sender() {
    return ex::just() | ex::then([] {
        int port = 8080;

        static_thread_pool pool{8};

        io::io_context ctx;
        set_sig_handler(ctx, SIGTERM);

        ex::sender auto snd = ex::on(ctx.get_scheduler(), listener(port, ctx, pool));
        ex::start_detached(std::move(snd));

        ctx.run();
        return 0;
    });
}
```

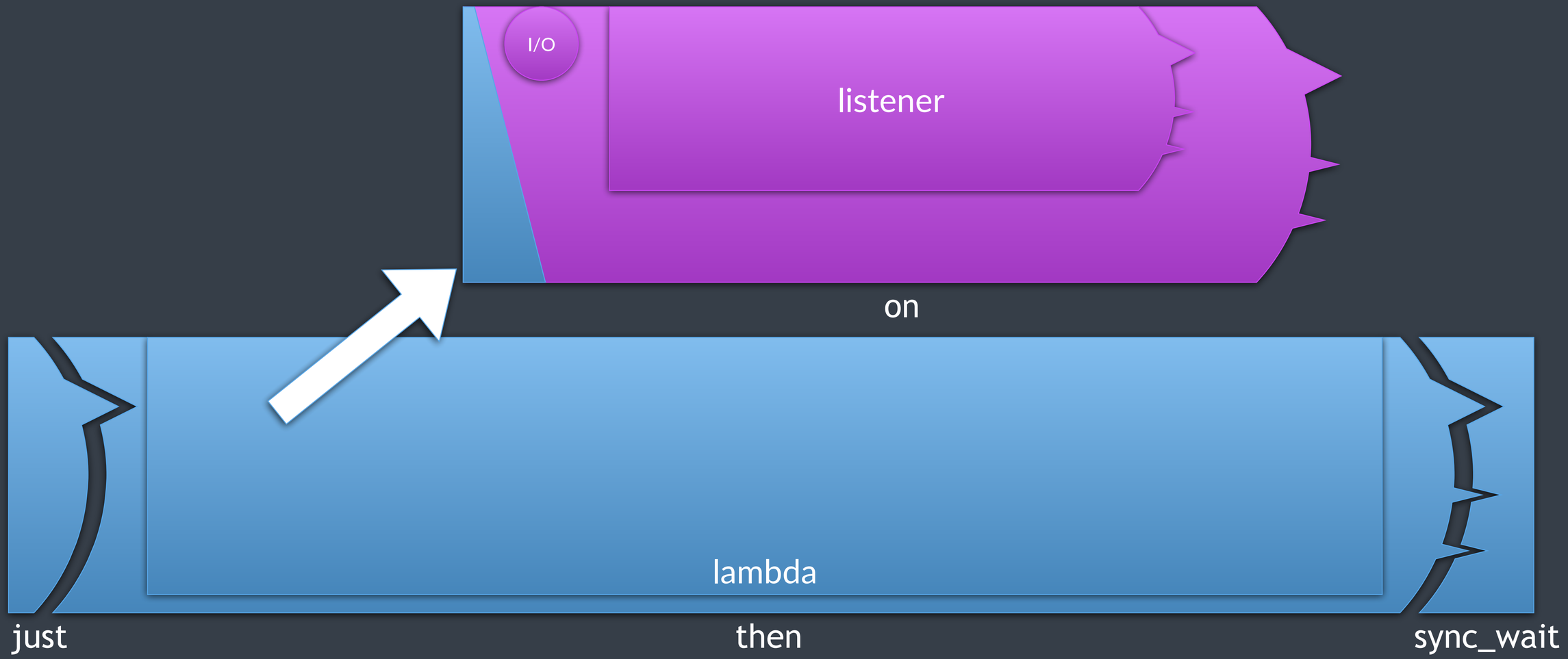
main()

I/O

CPU



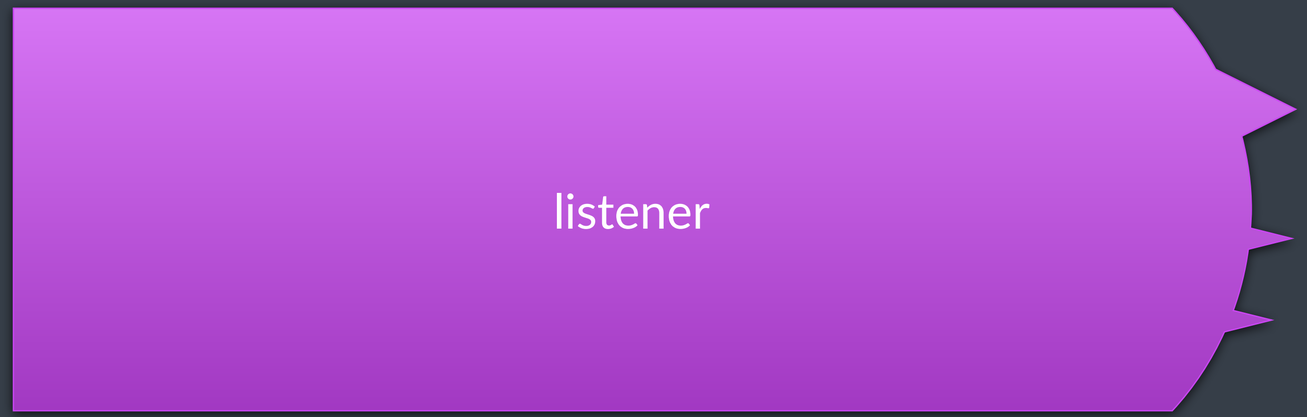
main()



listener

I/O

CPU

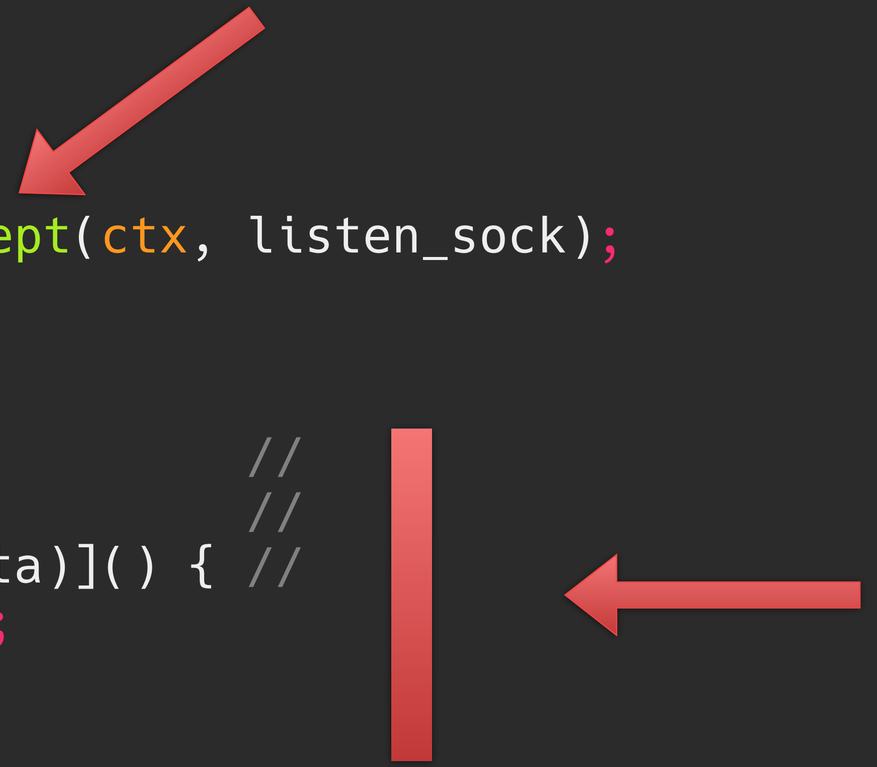


listener

```
auto listener(int port, io::io_context& ctx, static_thread_pool& pool)-> task<bool> {
    // ...
    co_return true;
}
```

listener

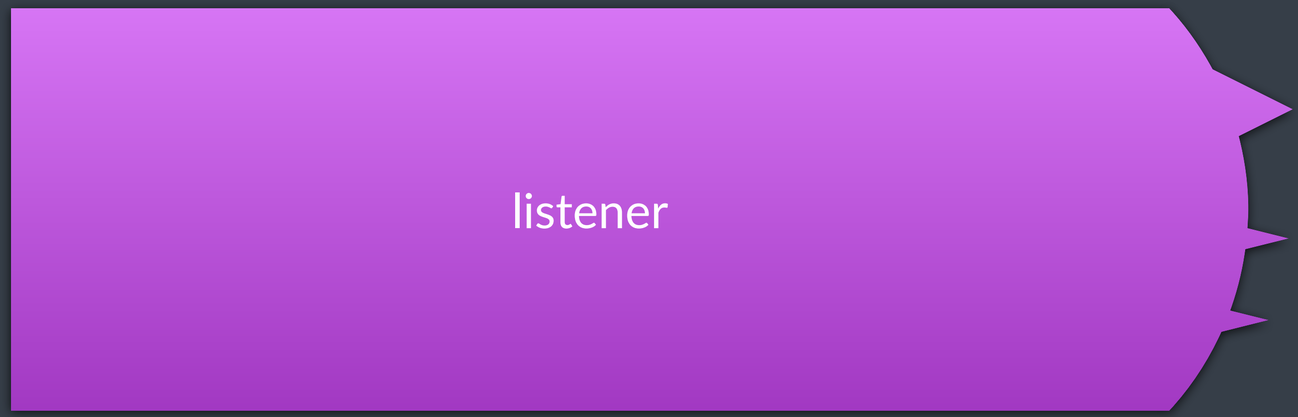
```
auto listener(int port, io::io_context& ctx, static_thread_pool& pool)-> task<bool> {
    io::listening_socket listen_sock;
    listen_sock.bind(port);
    listen_sock.listen();

    while (!ctx.is_stopped()) {
        io::connection conn = co_await io::async_accept(ctx, listen_sock);

        conn_data data{std::move(conn), ctx, pool};

        ex::sender auto snd =                                    //
                ex::just()                                       //
                | ex::let_value([data = std::move(data)]() {     //
                    return handle_connection(data);
                  });
        ex::start_detached(std::move(snd));
    }

    co_return true;
}
```



listener

I/O

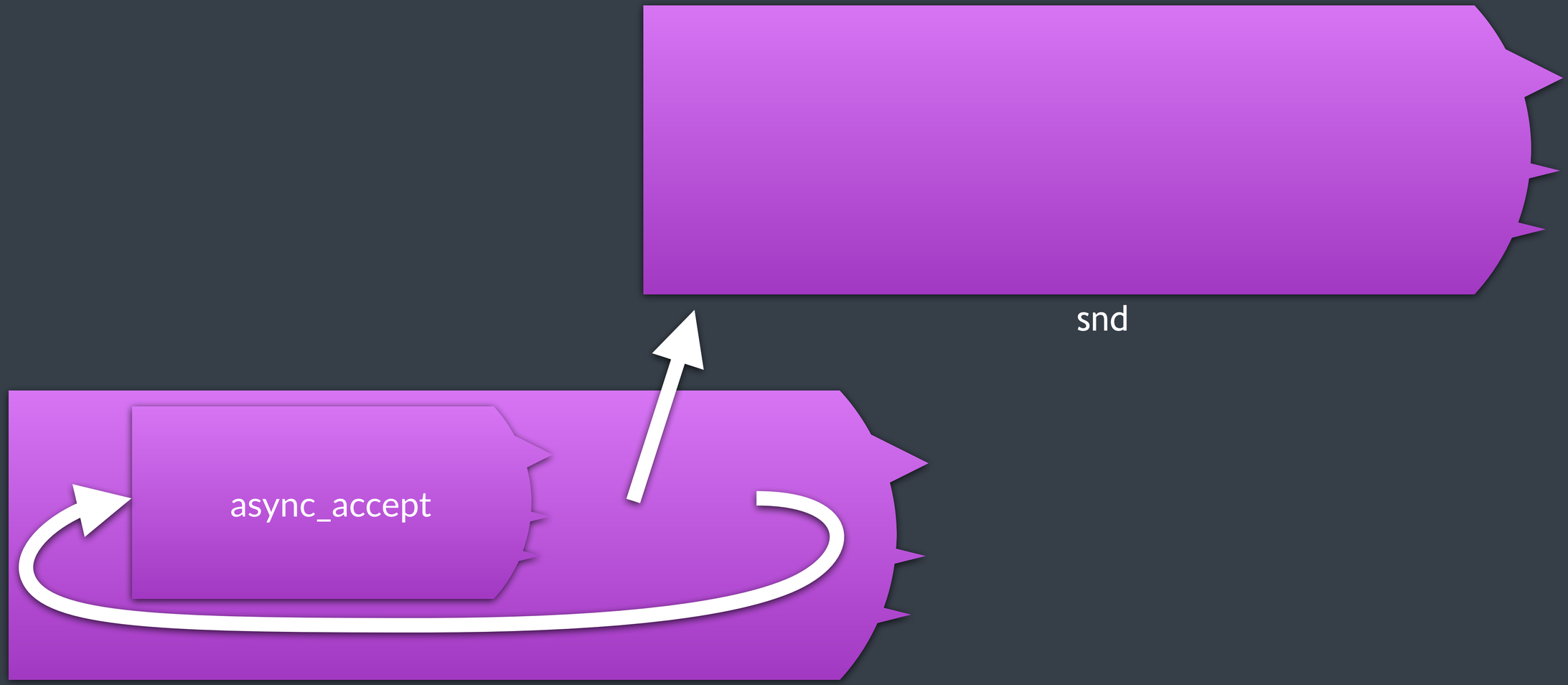
CPU



listener

I/O

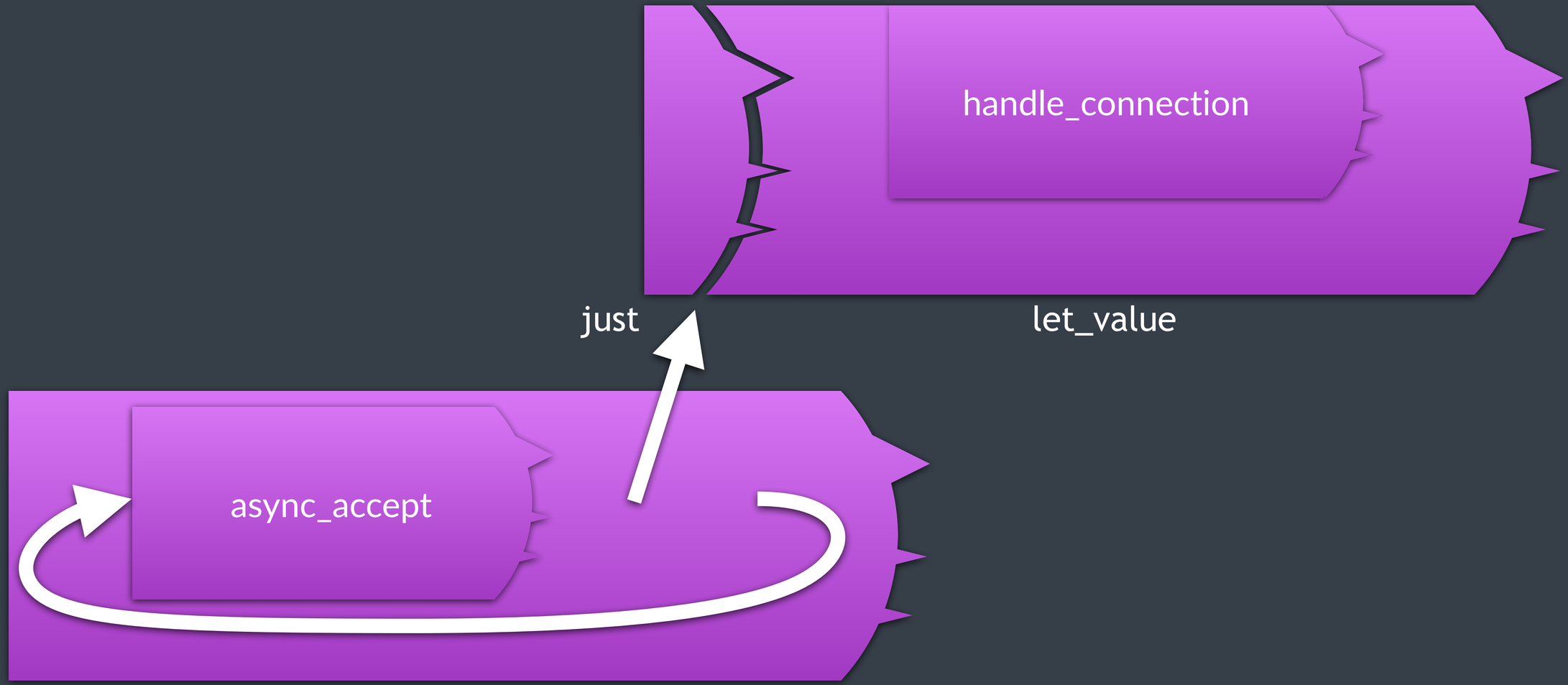
CPU



listener

I/O

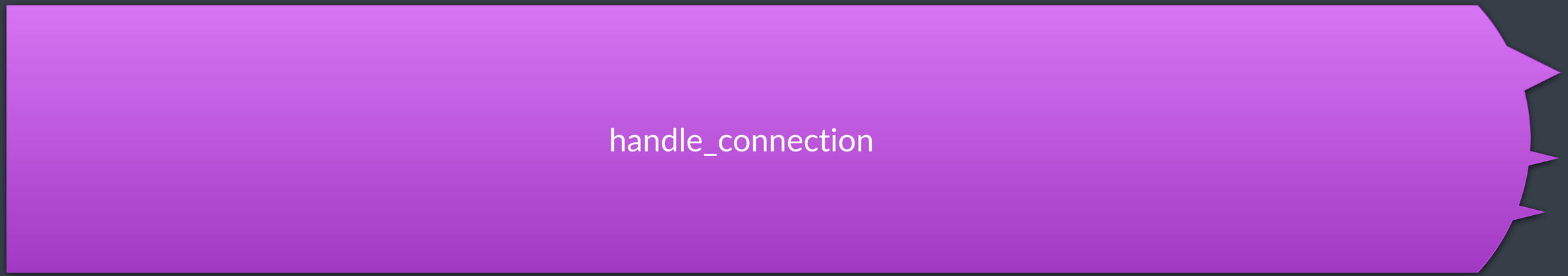
CPU



handle_connection

I/O

CPU



handle_connection

```
auto handle_connection(const conn_data& cdata) {
    return read_http_request(cdata.io_ctx_, cdata.conn_)
           | ex::transfer(cdata.pool_.get_scheduler())
           | ex::let_value([&cdata](http_server::http_request req) {
                return handle_request(cdata, std::move(req));
             })
           | ex::let_error([](std::exception_ptr) { return just_500_response(); })
           | ex::let_stopped([]() { return just_500_response(); })
           | ex::let_value([&cdata](http_server::http_response r) {
              return write_http_response(cdata.io_ctx_, cdata.conn_, std::move(r));
             });
}
```

handle_connection

just_500_response

```
auto just_500_response() {
    auto resp = http_server::create_response(
                    http_server::status_code::s_500_internal_server_error);
    return ex::just(std::move(resp));
}
```

read_http_request

```
auto read_http_request(io::io_context& ctx, const io::connection& conn)
        -> task<http_server::http_request> {
    http_server::request_parser parser;
    std::string buf;
    buf.reserve(1024 * 1024);
    io::out_buffer out_buf{buf};
    while (true) {
        std::size_t n = co_await io::async_read(ctx, conn, out_buf);
        auto data = std::string_view{buf.data(), n};
        auto r = parser.parse_next_packet(data);
        if (r)
            co_return {std::move(r.value())};
    }
}
```

read_http_request

I/O

CPU

write_http_response

```
auto write_http_response(io::io_context& ctx, const io::connection& conn,
        http_server::http_response resp) -> task<std::size_t> {
    std::vector<std::string_view> out_buffers;
    http_server::to_buffers(resp, out_buffers);
    std::size_t bytes_written{0};
    for (auto buf : out_buffers) {
        while (!buf.empty()) {
            auto n = co_await io::async_write(ctx, conn, buf);
            bytes_written += n;
            buf = buf.substr(n);
        }
    }
    co_return bytes_written;
}
```

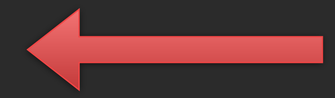
write_http_response

I/O

CPU

handle_request

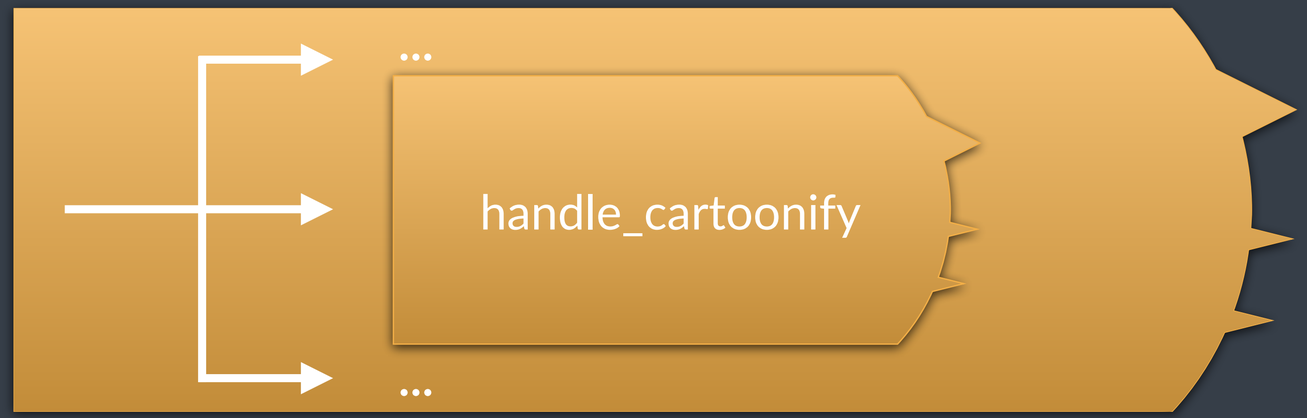
```
auto handle_request(const conn_data& cdata, http_server::http_request req)
        -> task<http_server::http_response> {
    auto puri = parse_uri(req.uri_);
    if (puri.path_ == "/transform/blur")
        co_return handle_blur(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/adaptthresh")
        co_return handle_adaptthresh(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/reducecolors")
        co_return handle_reducecolors(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/cartoonify")
        co_return co_await handle_cartoonify(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/oilpainting")
        co_return handle_oilpainting(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/contourpaint")
        co_return co_await handle_contourpaint(cdata, std::move(req), puri);
   co_return http_server::create_response(http_server::status_code::s_404_not_found);
}
```



handle_request

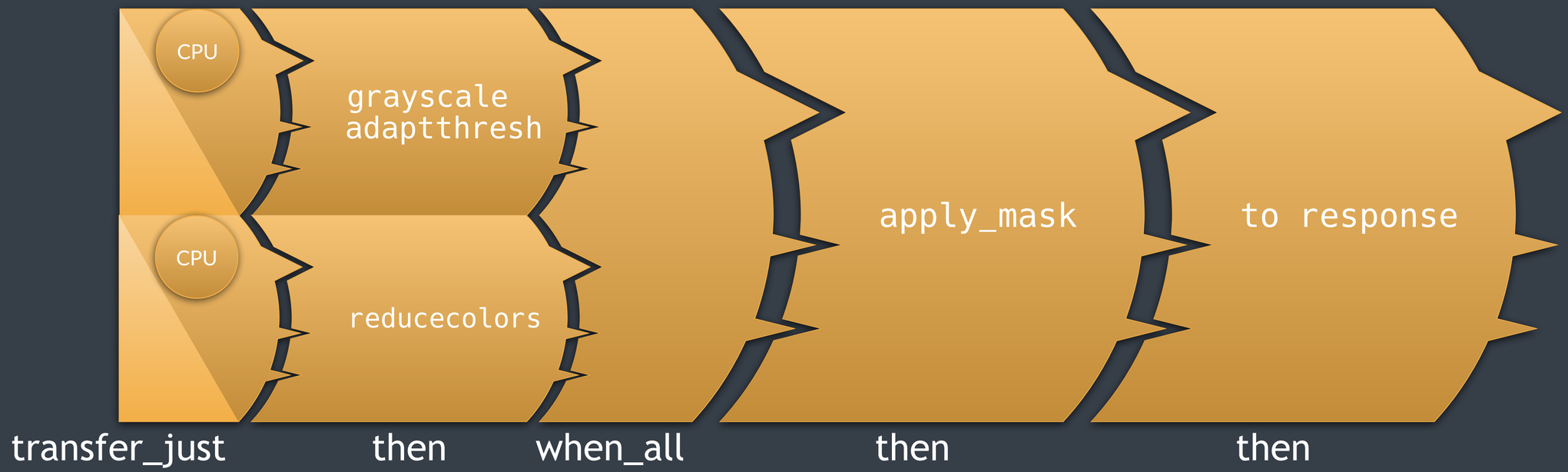
I/O

CPU



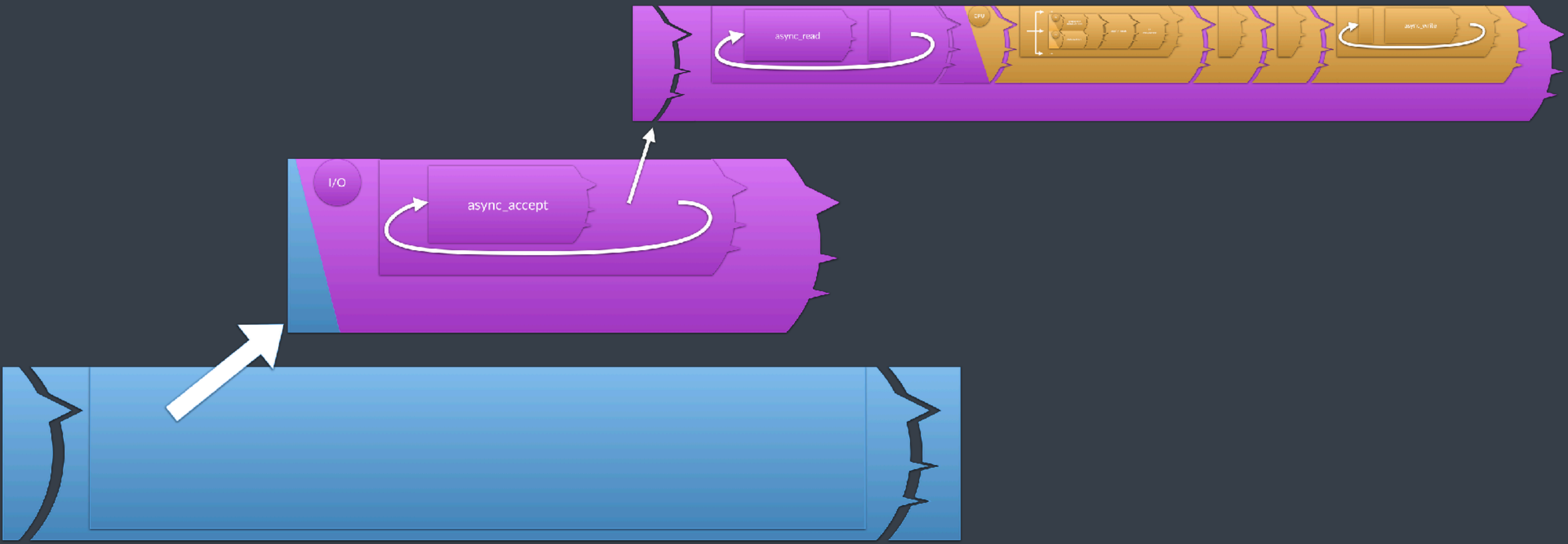
```
auto handle_cartoonify(const conn_data& cdata, http_server::http_request&& req, parsed_uri puri)
        -> task<http_server::http_response> {
    int blur_size = get_param_int(puri, "blur_size", 3);
    int num_colors = get_param_int(puri, "num_colors", 5);
    int block_size = get_param_int(puri, "block_size", 5);
    int diff = get_param_int(puri, "diff", 5);
    auto src = to_cv(req.body_);
    ex::sender auto snd = ex::when_all(
        ex::transfer_just(cdata.pool_.get_scheduler(), src)
            | ex::then([=](const cv::Mat& src) {
                auto gray = tr_to_grayscale(tr_blur(src, blur_size));
                return tr_adaptthresh(gray, block_size, diff);
              }),
        ex::transfer_just(cdata.pool_.get_scheduler(), src)
                | ex::then([=](const cv::Mat& src) {
                    return tr_reducecolors(src, num_colors);
                  })
        )
      | ex::then([](const cv::Mat& edges, const cv::Mat& reduced_colors) {
           return tr_apply_mask(reduced_colors, edges);
        })
      | ex::then(img_to_response);
    co_return co_await std::move(snd);
}
```

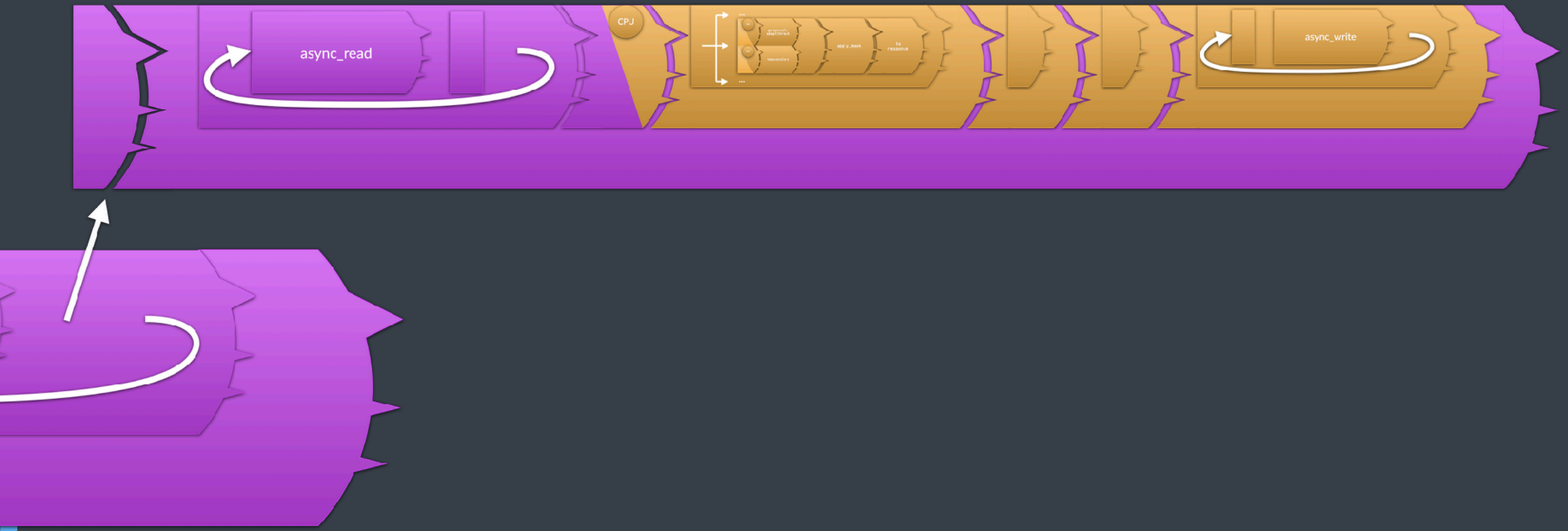
handle_cartoonify

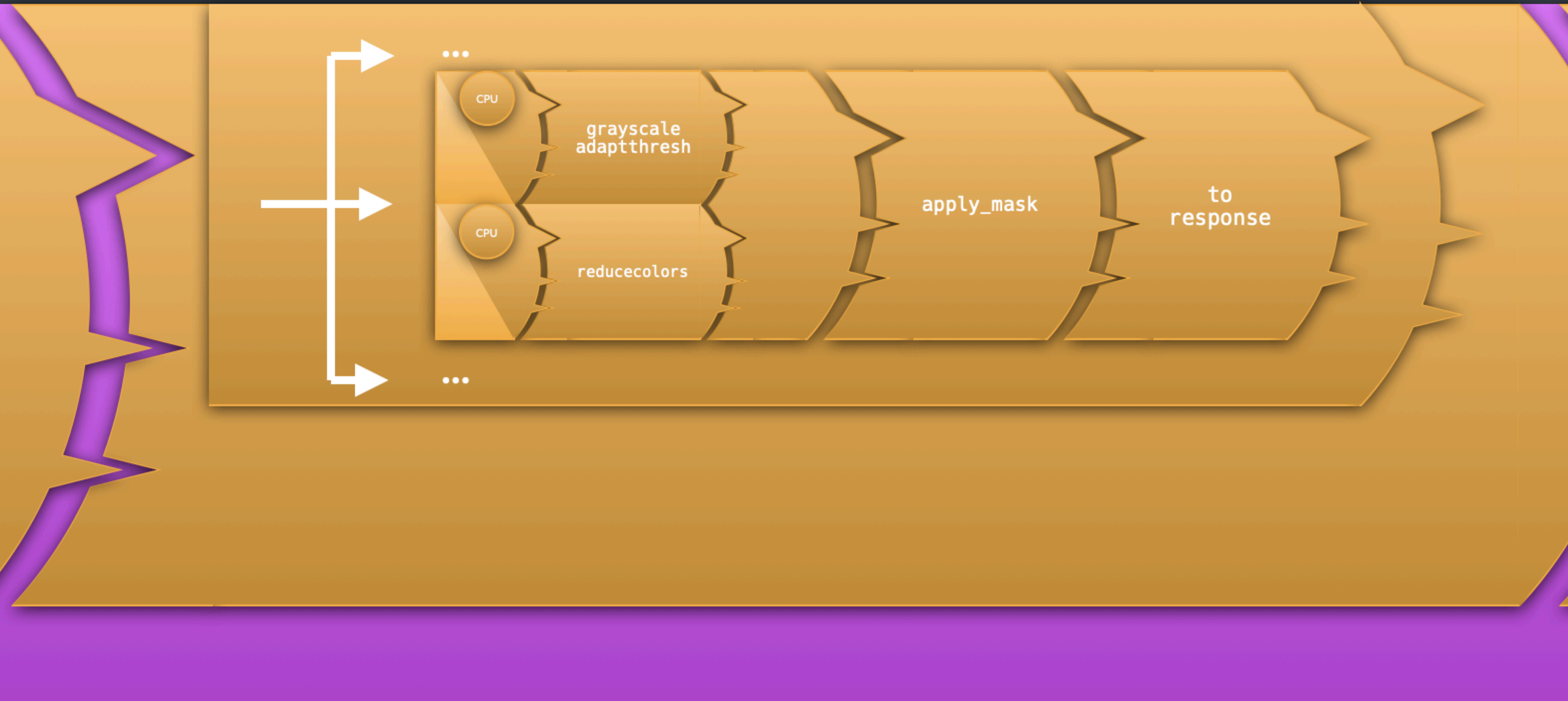


recursive decomposition

- get_main_sender
  - listener
    - async_accept
    - handle_connection
      - read_http_request
        - async_read
      - handle_request
        - handle_cartoonify
        - …
      - just_500_response
      - write_http_response
        - async_write


async_read
async_write
I/O
async_accept


async_read
CPJ
async_write


...
CPU
grayscale
adaptthresh
CPU
reducecolors
apply_mask
to
response
...

Conclusions

6


DIFFICULT
ROADS
LEAD TO
BEAUTIFUL
DESTINATIONS

senders

good abstractions for concurrency
no need for synchronization
highly composable

a pattern language

shapes of sender algorithms -> basic patterns

patterns -> lexicon

nuances for patterns

composing patterns -> syntax

speaking **concurrency** ?

a language for **safety**
a language for **efficiency**
structured language

Thank You

@LucT3o

lucteo.ro