Structured Concurrency in Hylo

LUCIAN RADU TEODORESCU
GARMIN

poll

why threads?


speed

poll

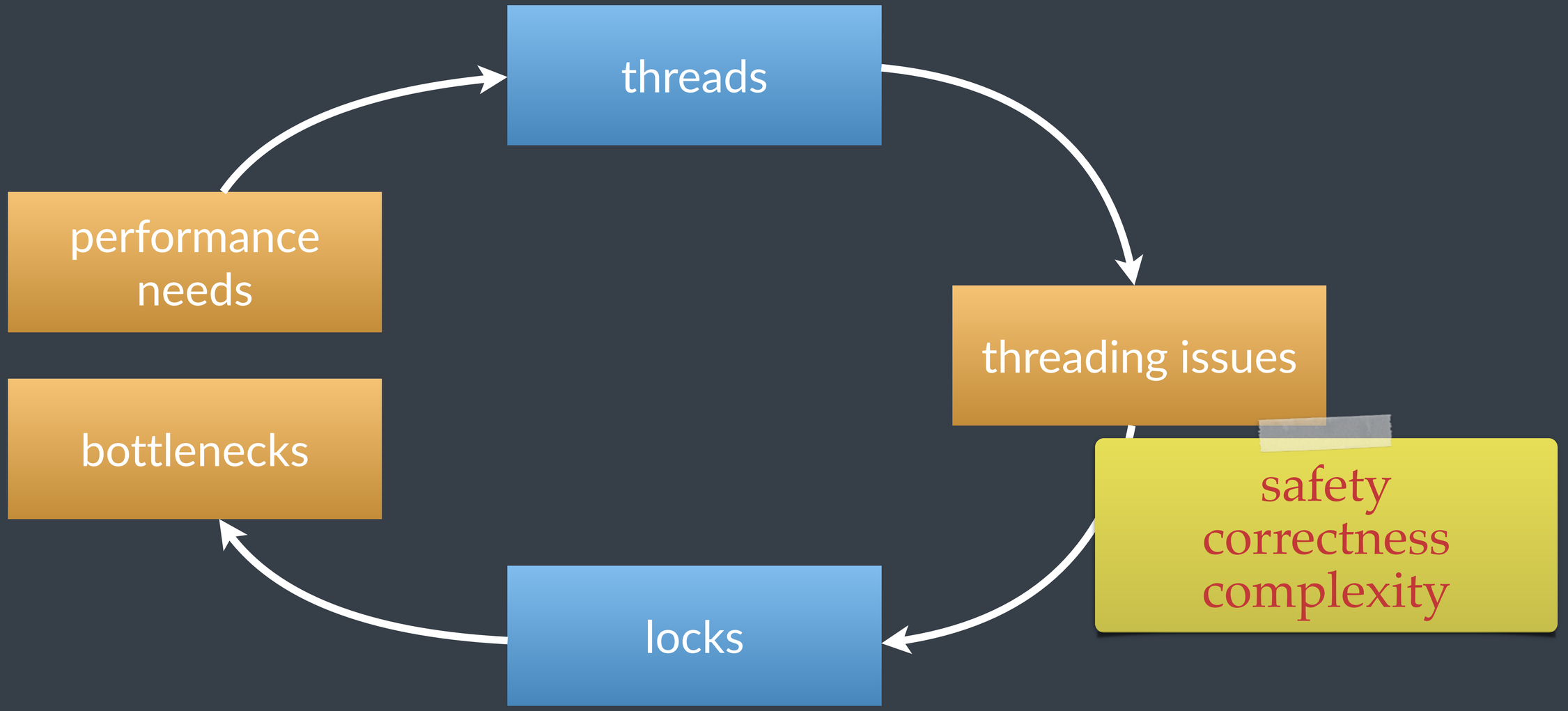
typical problems?

poll

typical problems?
typical solutions?

locks

locks are **bottlenecks**


threads
threading issues
safety
correctness
complexity
locks
bottlenecks
performance
needs

goals

concurrent code ~ sequential code

“structured programming”, but for concurrency

make concurrency reasonable

work in progress

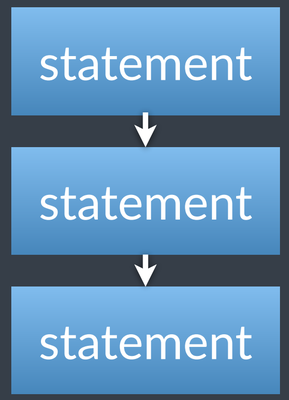
1. Concurrent programming
2. Hylo
3. Expressing concurrency
4. Implementation details
5. Asynchrony
6. Analysis
7. Conclusions

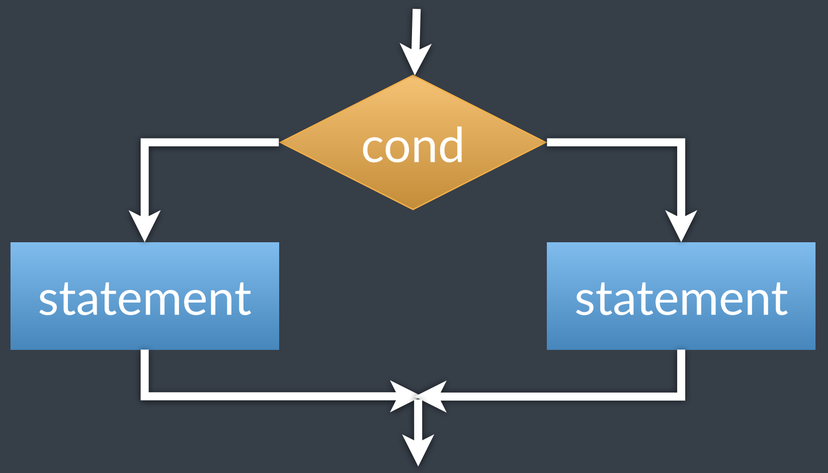
lucteo.ro/pres/2024-icooolps

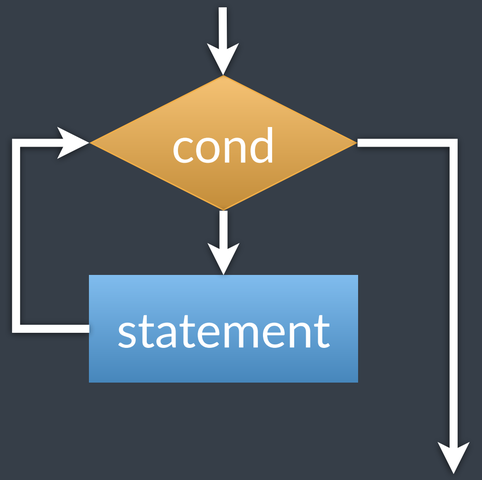
Concurrent programming

operations

a program is a set of interconnected **operations**







D. G. BOBROW, Editor

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

CORRADO BÖHM AND GIUSEPPE JACOPINI
International Computation Centre and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent *inter alia* mappings of a set into itself. Although not every diagram is decomposable into a finite number of given base diagrams, this becomes true at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing maching provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by composition and iteration from the two machines λ and R. That family is a proper subfamily of the whole family of Turing machines.

1. Introduction and Summary

The set of block or flow diagrams is a two-dimensional programming language, which was used at the beginning of automatic computing and which now still enjoys a certain favor. As far as is known, a systematic theory of this language does not exist. At the most, there are some papers by Peter [1], Gorn [2], Hermes [3], Ciampa [4], Riguet [5], Ianov [6], Asser [7], where flow diagrams are introduced with different purposes and defined in connection with the descriptions of algorithms or programs.

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow daigram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional type

This paper was presented as an invited talk at the 1964 International Colloquium on Algebraic Linguistics and Automata Theory, Jerusalem, Israel. Preparation of the manuscript was supported by National Science Foundation Grant GP-2380.

This work was carried out at the Istituto Nazionale per le Applicazioni del Calcolo (INAC), in collaboration with the International Computation Centre (ICC), under the Italian Consiglio Nazionale delle Ricerche (CNR) Research Group No. 22 for 1963–64.

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which

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

1. use of abstractions

abstractions can be used as operations

2. recursive decomposition

a program can be recursively decomposed into operations
(operations nest)

3. regular shape

operations have one entry point, and one exit point

4. local reasoning

within a scope, one can arrange the operations in a way that enables local reasoning

5. soundness and completeness

all programs can be written in a way that respect the above laws

concurrency

happens-before

$a < b$

sequential program

total ordering on operation execution

concurrent program

partial ordering on operation execution

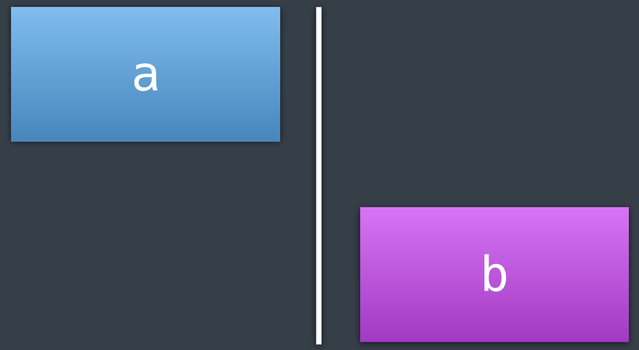
modeling concurrency

at runtime

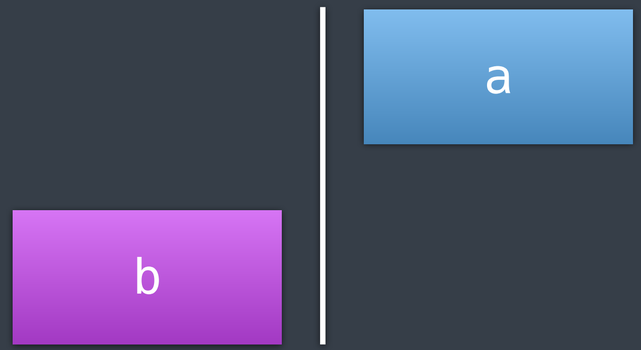
3 execution possibilities

$$a < b$$
$$b < a$$
$$\neg(a < b) \land \neg(b < a)$$

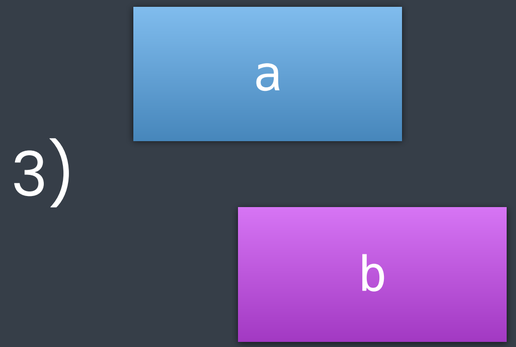
1)



2)



3)



concurrency
(design time)

expressing execution constraints
ignoring actual execution

design time

basic concurrent constraints

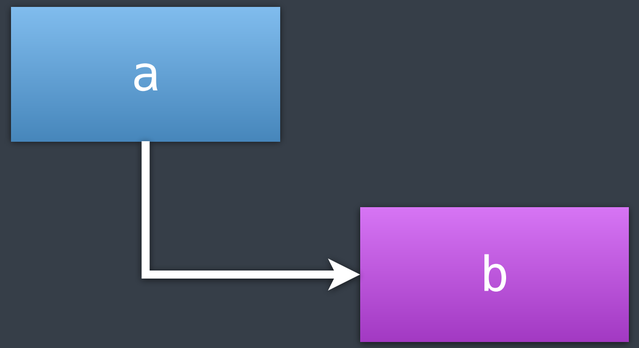
$$a < b$$

$$b < a$$

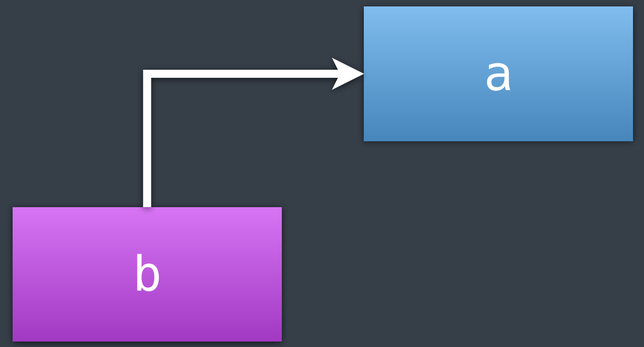
$$(a < b) \lor (b < a)$$ mutual exclusion

$$\neg(a < b) \land \neg(b < a)$$ concurrent execution

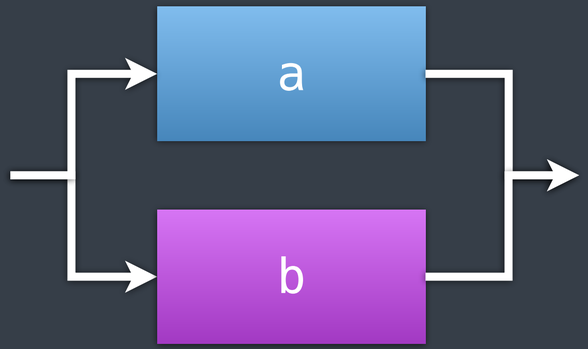
design time



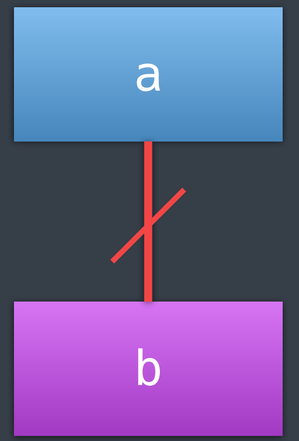
a < b



b < a



concurrent execution



mutual execution

advanced concurrent constraints

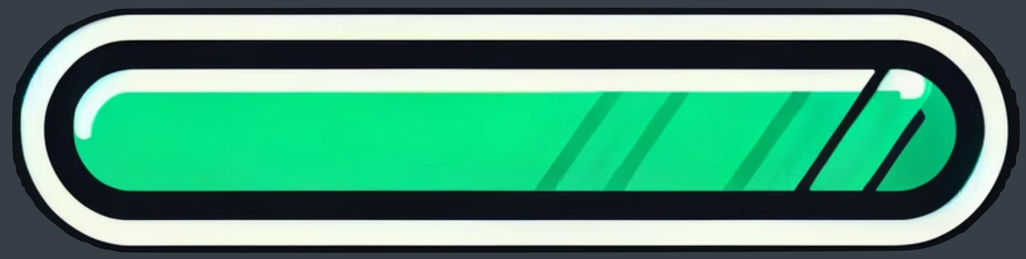
conditional concurrency

(sometimes exclusion, sometimes concurrent)

more than 2 operations

focus

local constraints
non-dynamic constraints



100%

there is nothing more to concurrency

structured concurrency

concurrency doesn't contradict **structured programming**

implementation notes

thread conservation

one thread in, one thread out

stack conservation

one in stack, one out stack

(out stack >= in stack)

thread-stack relation

one thread, one stack

number of active threads

the number of stacks that can vary at the same time

logical thread

variation of stack pointer over time


abstract out the OS threads

Hylo

2

Hylo programming language

fast by definition
safe by default
simple

www.hylo-lang.org

builds upon the best parts from C++

value semantics

pass by value, without copy

copies & moves are explicit

consuming move semantics

rules for capture access w/o consuming

Mutable Value Semantics

Swift

w/o reference semantics

Rust

w/o lifetime annotations

functional

with controlled mutation

value semantics

```
template <typename T>
void append2(std::vector<T>& destination, const T& value) {
  destination.push_back(value);
  destination.push_back(value);
}

std::vector<int> data;
…
append2(data, data[0]);
```

value semantics


```
fun append2<T>(_ destination: inout Array<T>, _ value: T) {
  &destination.push_back(value)
  &destination.push_back(value)
}

var data: Array<Int>
…
append2(&data, data[0])      // ERROR
let value = data[0].copy()
append2(&data, value)        // OK
```

copies & moves
are explicit

law of exclusivity

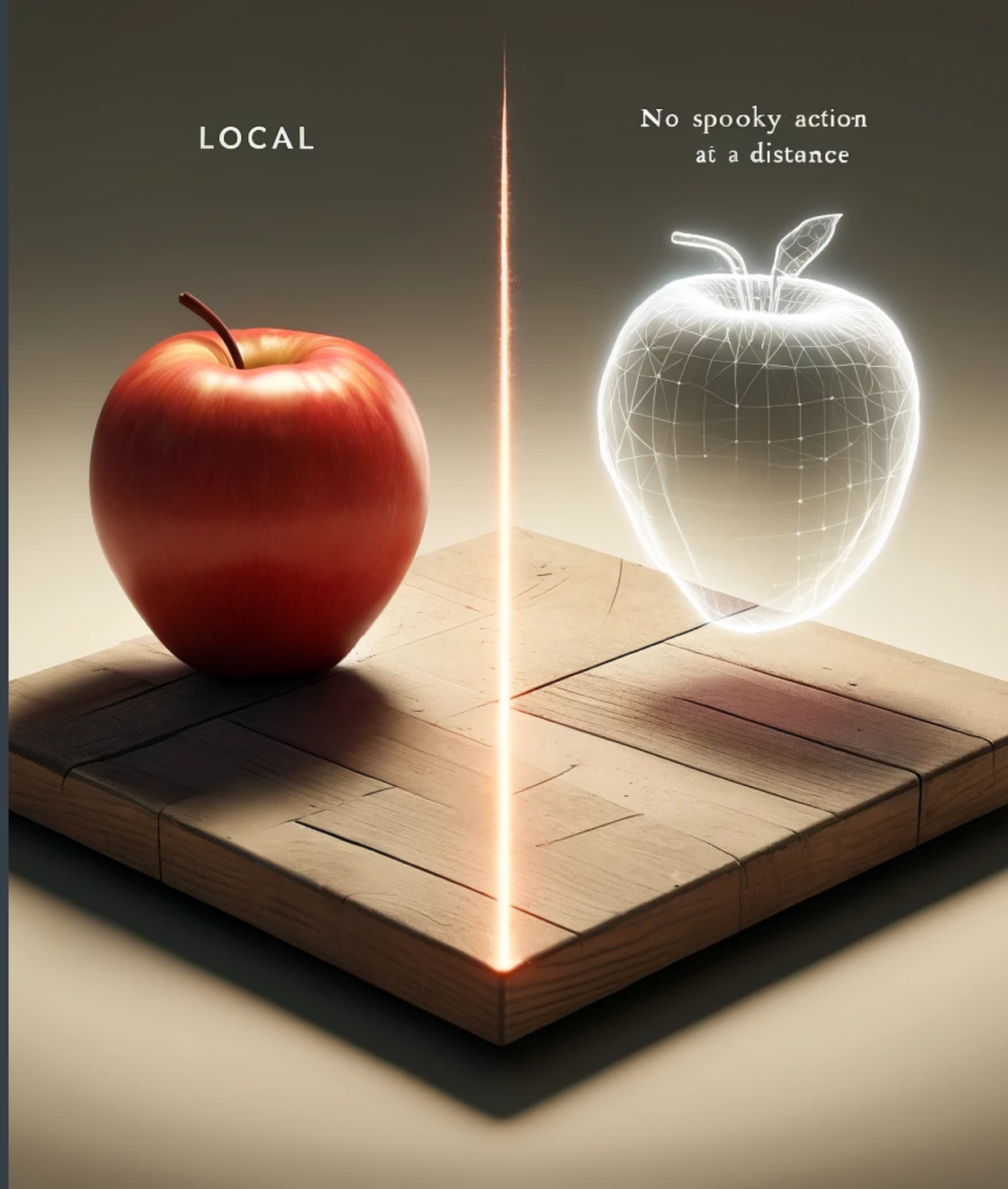
no simultaneous **read + write** access

no simultaneous **write + write** access

read + read = ok

local reasoning

no spooky action at a distance



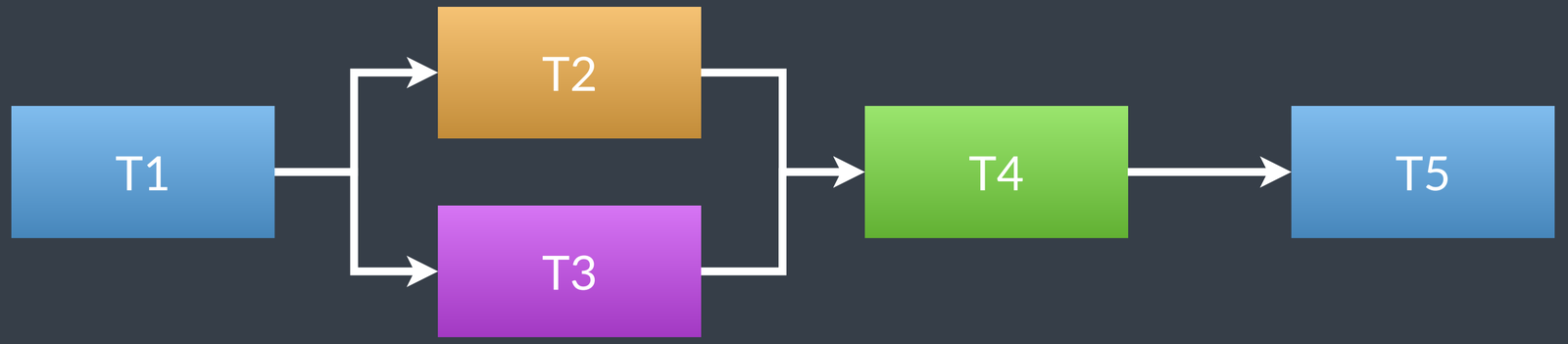
Expressing concurrency

3

Hello, concurrent world!

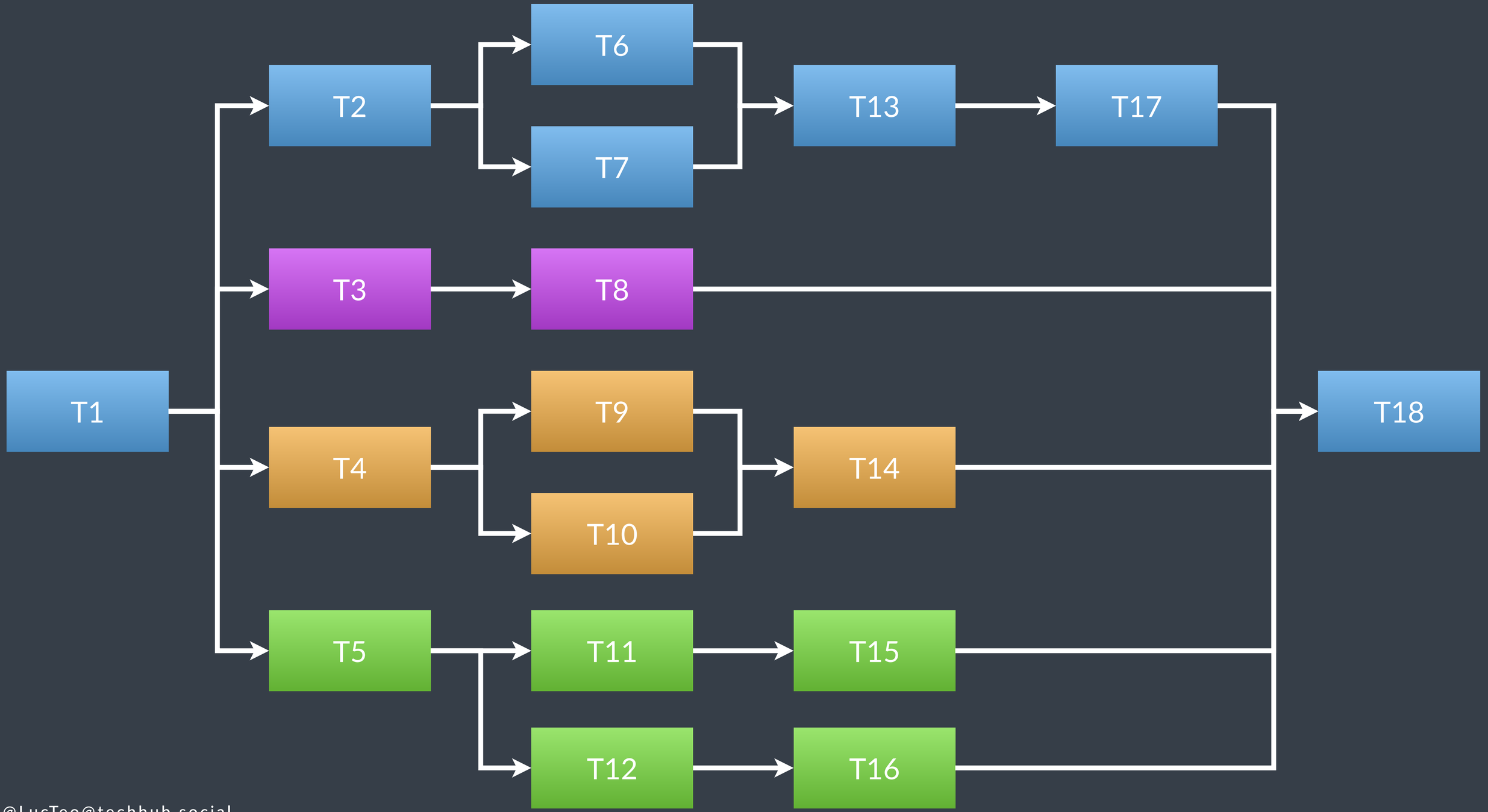

```
fun concurrent_greeting() {
  var f = spawn {
    print("Hello, concurrent world!")
  }
  // do some other things...
  f.await()
}
```

task relations

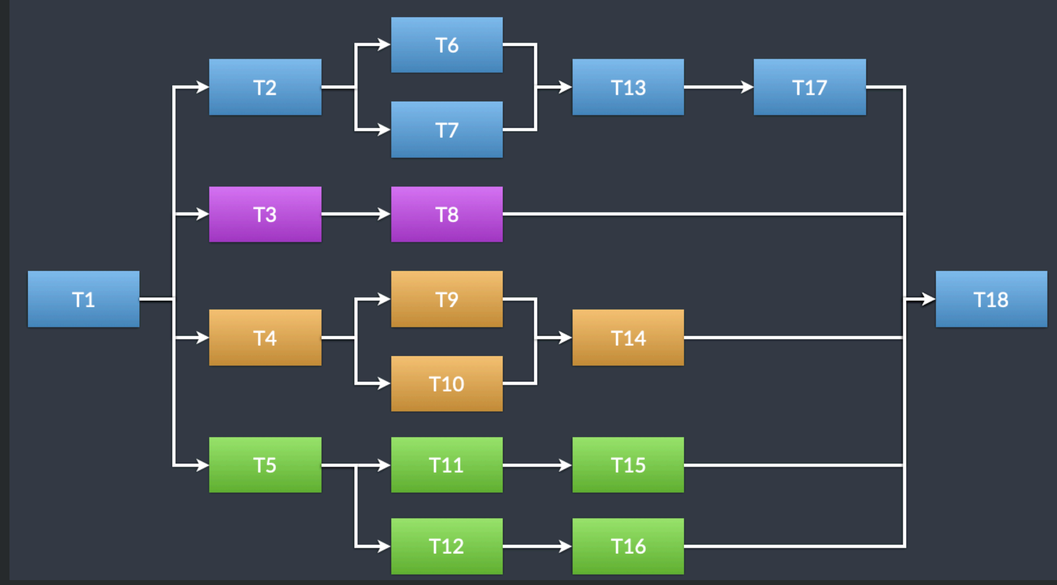


task relations

```
fun task_relations() {
  print("T1")
  var f = spawn { print("T3") }
  print("T2")
  f.await()
  print("T4")
  print("T5")
}
```


T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T11
T12
T13
T14
T15
T16
T17
T18

```
fun run_work() -> Int {
  var sum = 0
  &sum += run_task(1)

  var f2 = spawn_(fun[] () -> Int {
    var local_sum = 0
    &local_sum += run_task(2)

    var f = spawn_(fun[] () -> Int { return run_task(7) })
    &local_sum += run_task(6)
    &local_sum += f.await()

    &local_sum += run_task(13)
    &local_sum += run_task(17)
    return local_sum
  })

  var f3 = spawn_(fun[] () -> Int {
    return run_task(3) + run_task(8)
  })

  var f4 = spawn_(fun[] () -> Int {
    var local_sum = 0
    &local_sum += run_task(4)

    var f = spawn_(fun[] () -> Int { return run_task(10) })
    &local_sum += run_task(9)
    &local_sum += f.await()

    &local_sum += run_task(14)
    return local_sum
  })

  var f5 = spawn_(fun[] () -> Int {
    var local_sum = 0
    &local_sum += run_task(5)

    var f = spawn_(fun[] () -> Int { return run_task(12) + run_task(16) })
    &local_sum += run_task(11) + run_task(15)
    &local_sum += f.await()

    return local_sum
  })

  sum += f2.await() + f3.await() + f4.await() + f5.await()
  &sum += run_task(18)
  return sum
}
```



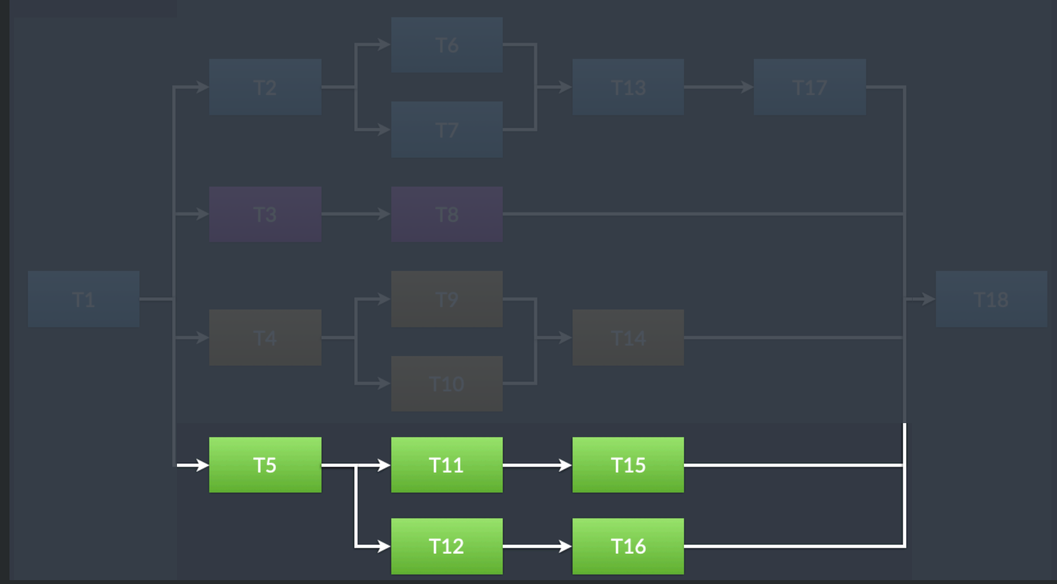
@LucTeo@techhub.social


```
fun run_work() -> Int {
  var sum = 0
  &sum += run_task(1)

  var f2 = …
  var f3 = …
  var f4 = …
  var f5 = …

  sum += f2.await() + f3.await() + f4.await() + f5.await()
  &sum += run_task(18)
  return sum
}
```


```
var f2 = spawn_(fun[] () -> Int {
  var local_sum = 0
  &local_sum += run_task(2)

  var f = spawn_(fun[] () -> Int { return run_task(7) })
  &local_sum += run_task(6)
  &local_sum += f.await()

  &local_sum += run_task(13)
  &local_sum += run_task(17)
  return local_sum
})
```


```
var f3 = spawn_(fun[] () -> Int {
  return run_task(3) + run_task(8)
})
```


```
var f4 = spawn_(fun[] () -> Int {
  var local_sum = 0
  &local_sum += run_task(4)

  var f = spawn_(fun[] () -> Int { return run_task(10) })
  &local_sum += run_task(9)
  &local_sum += f.await()

  &local_sum += run_task(14)
  return local_sum
})
```




```
var f5 = spawn_(fun[] () -> Int {
  var local_sum = 0
  &local_sum += run_task(5)

  var f = spawn_(fun[] () -> Int { return run_task(12) + run_task(16) })
  &local_sum += run_task(11) + run_task(15)
  &local_sum += f.await()

  return local_sum
})
```

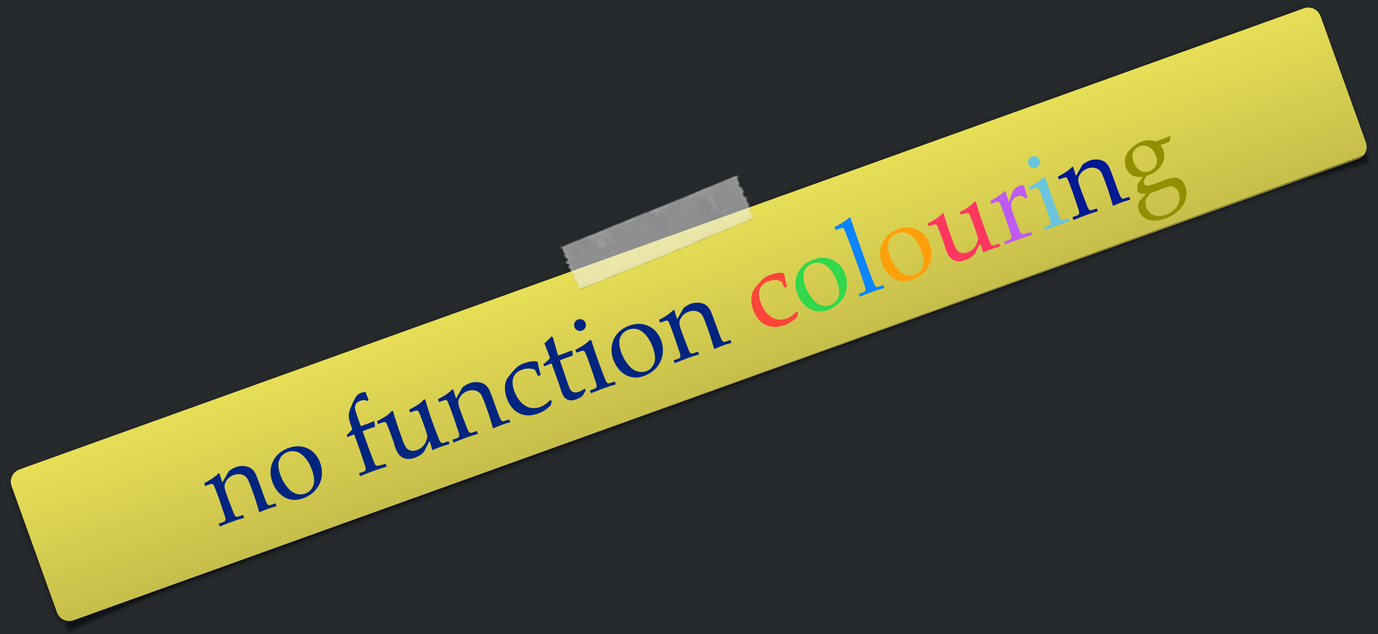
async/await model

async vs sequential

```
fun concurrent_greeting() {
  var f = spawn {
    print("Hello, concurrent world!")
  }
  // do some other things...
  f.await()
}


fun regular_greeting() {
  print("Hello, sequential world!")
  // do some other things...
}
```

async vs sequential

```
fun concurrent_greeting() {
  var f = spawn {
    print("Hello, concurrent world!")
  }
  // do some other things...
  f.await()
}

fun regular_greeting() {
  print("Hello, sequential world!")
  // do some other things...
}
```



spawn/await = operation

```
fun example() {
  A()

  var f = spawn { C() }
  B()
  f.await()

  D()
}
```

spawn/await = operation

```
fun example() {
  A()

  B_and_C()

  D()
}
```

```
fun B_and_C() {
  var f = spawn { C() }
  B()
  f.await()
}
```

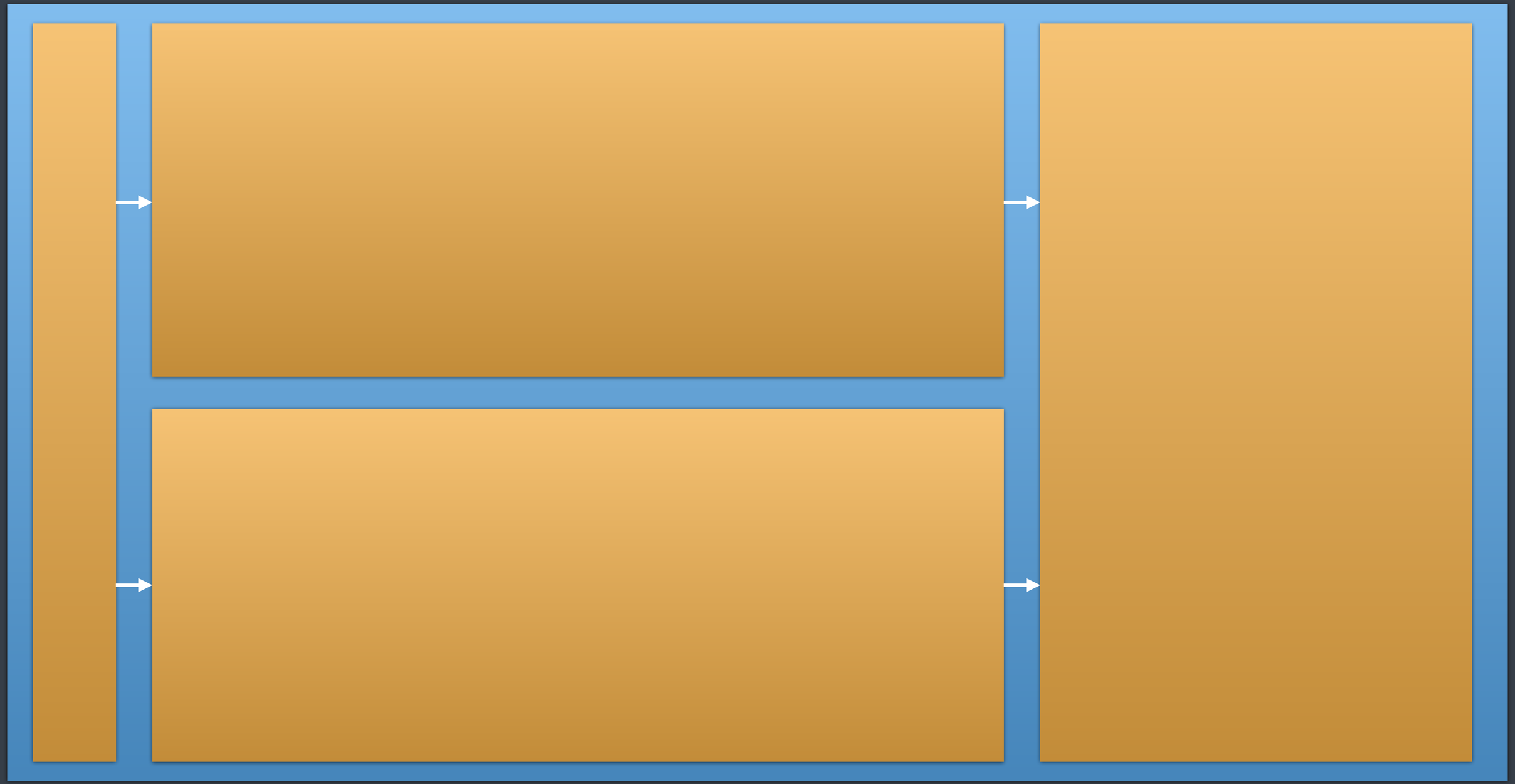
1. use of abstractions

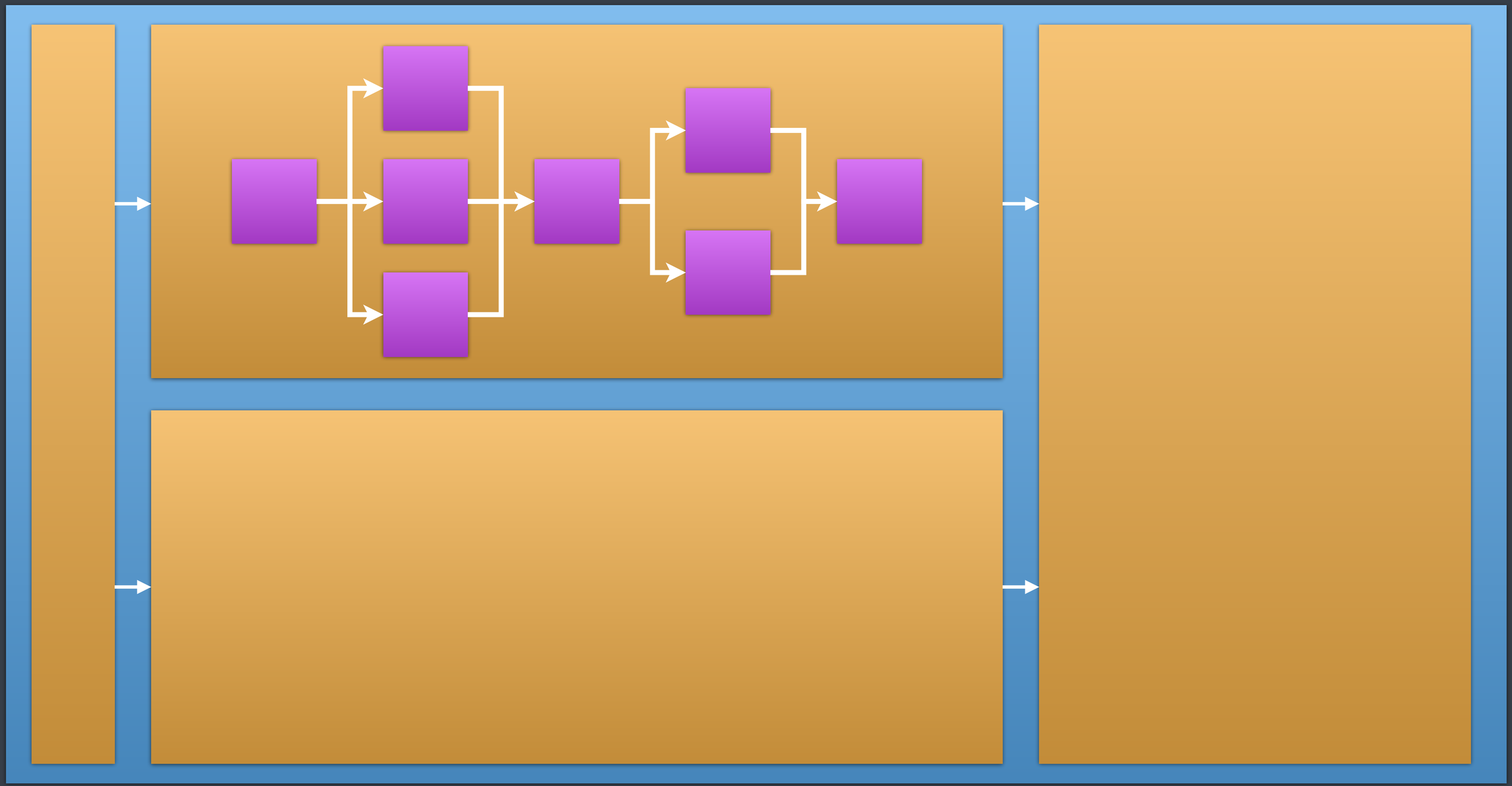
functions can encapsulate concurrency

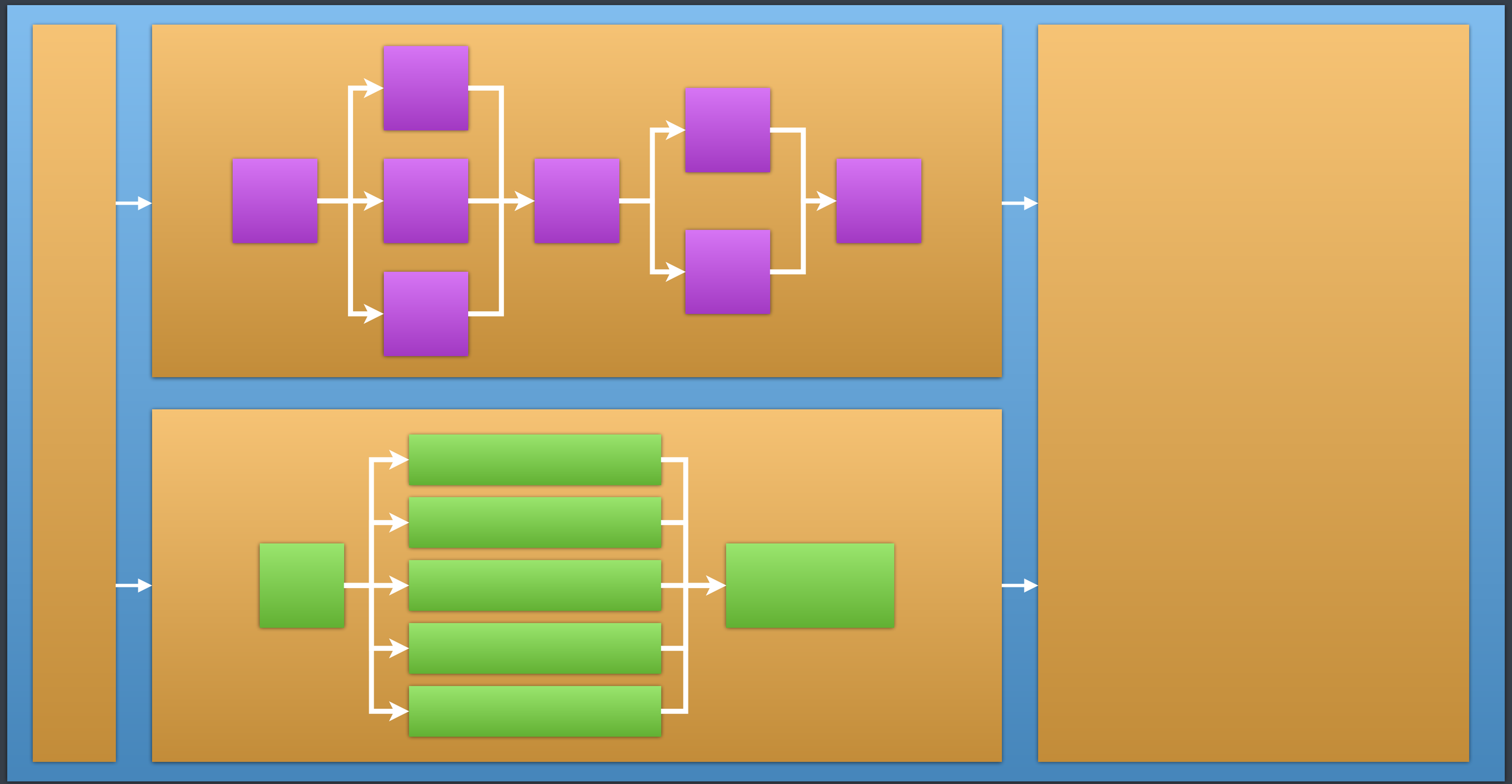
2. recursive decomposition

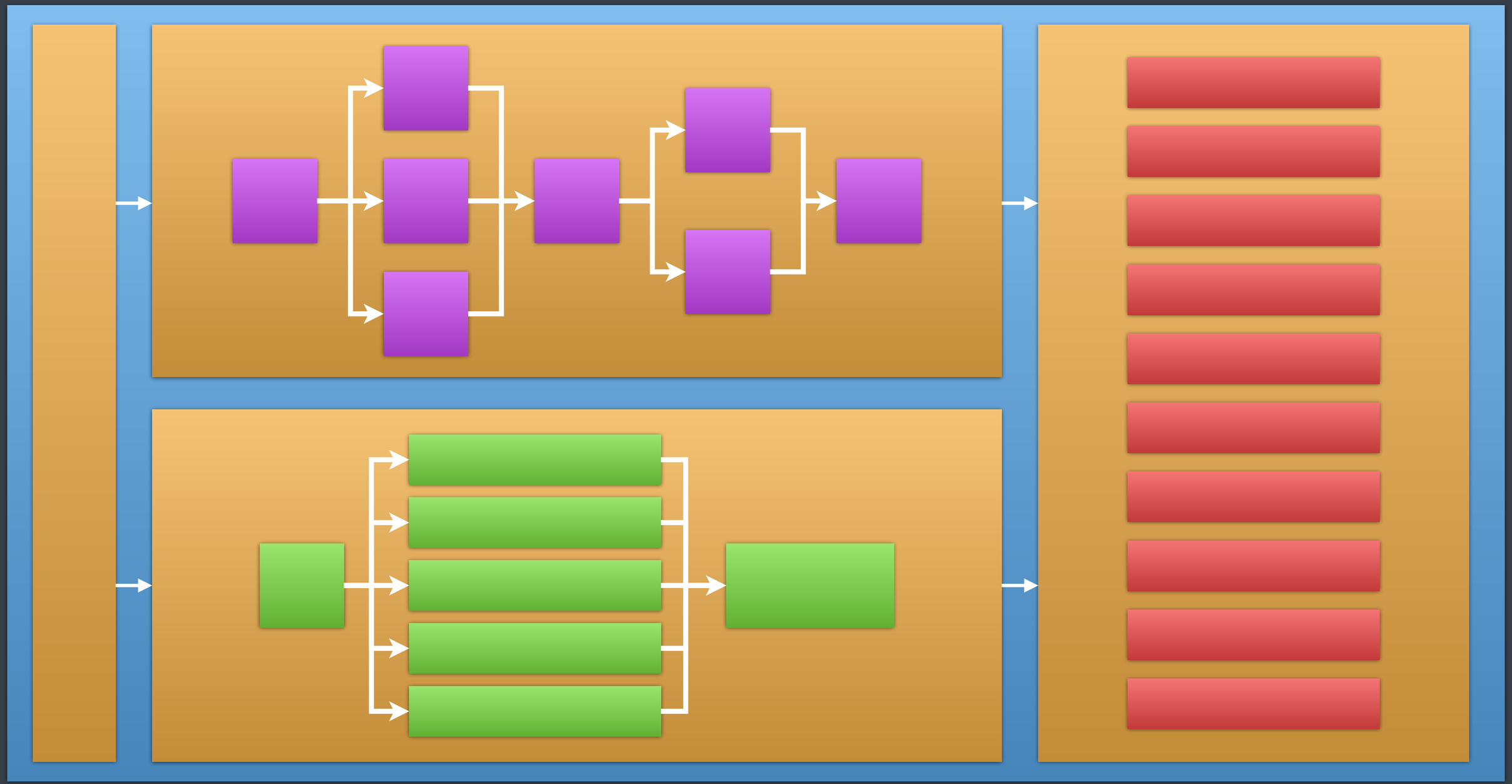
decomposition through functions

program









recursive decomposition

functions all the way down

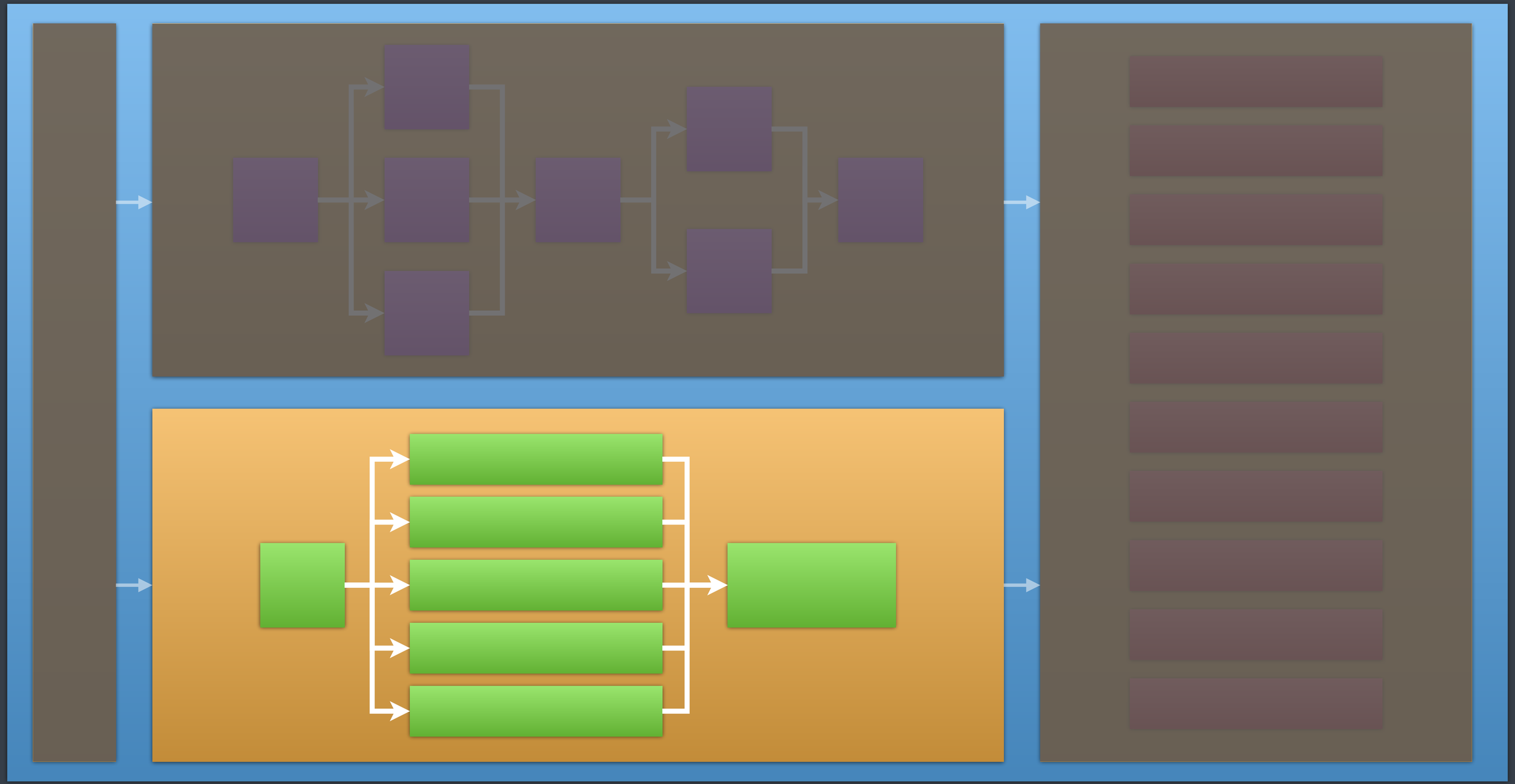
3. regular shape

one entry **point**, and one exit **point**
one entry **thread**, one exit **thread**
one entry **stack**, one exit **stack**

4. local reasoning

same as with function calls

Hylo ❤️ local reasoning



reasonable concurrency

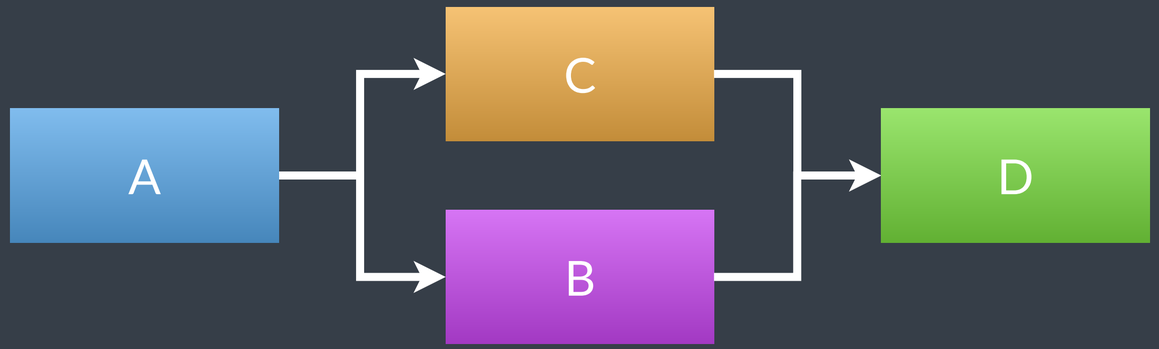
5. soundness and completeness

same as in structured programming

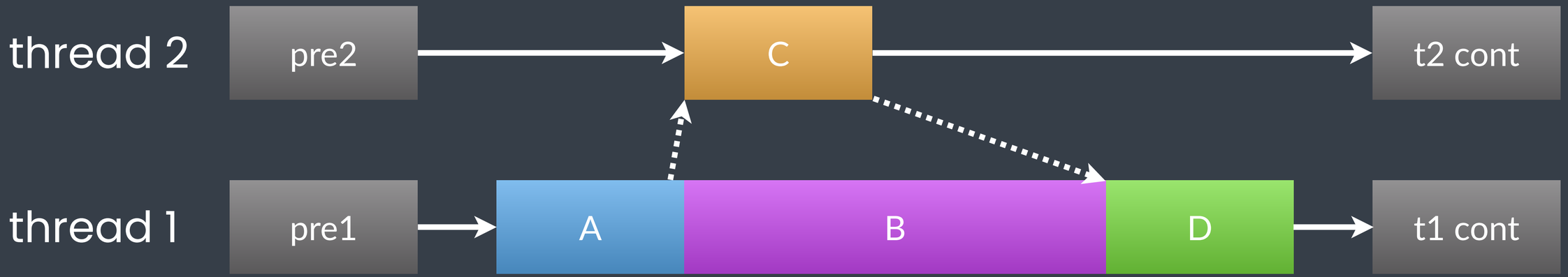
Implementation details

4

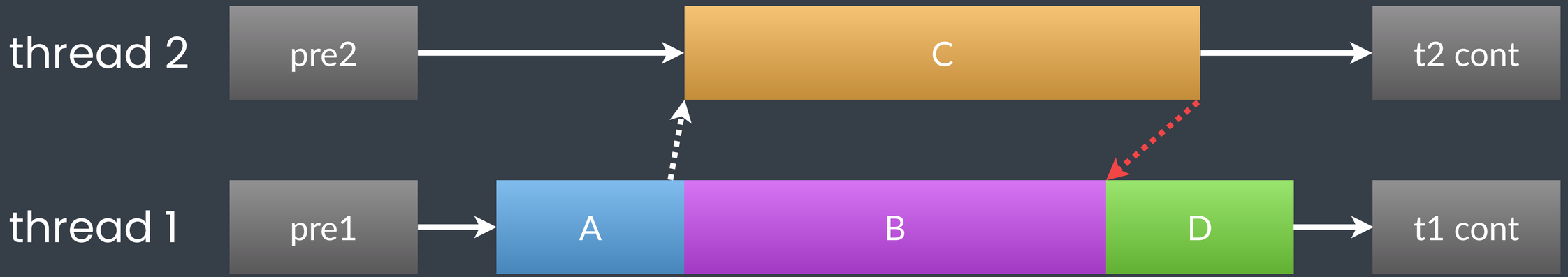
```
fun example() {
  A()

  var f = spawn { C() }
  B()
  f.await()

  D()
}
```



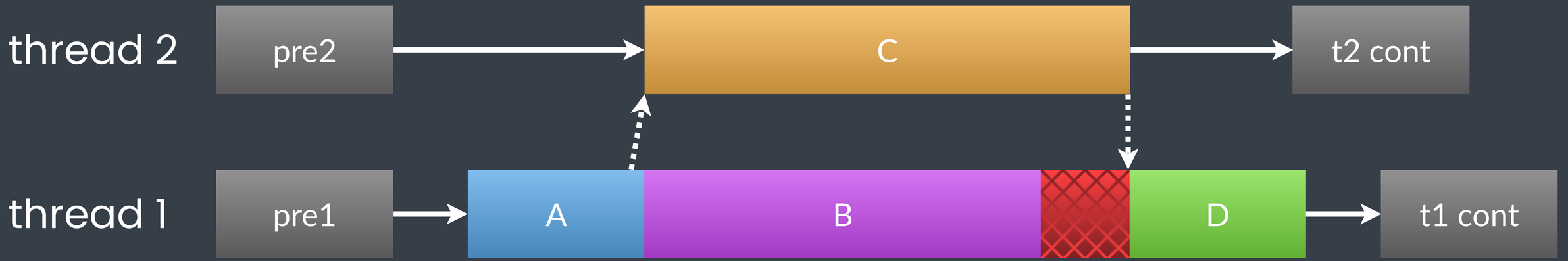
concurrency design



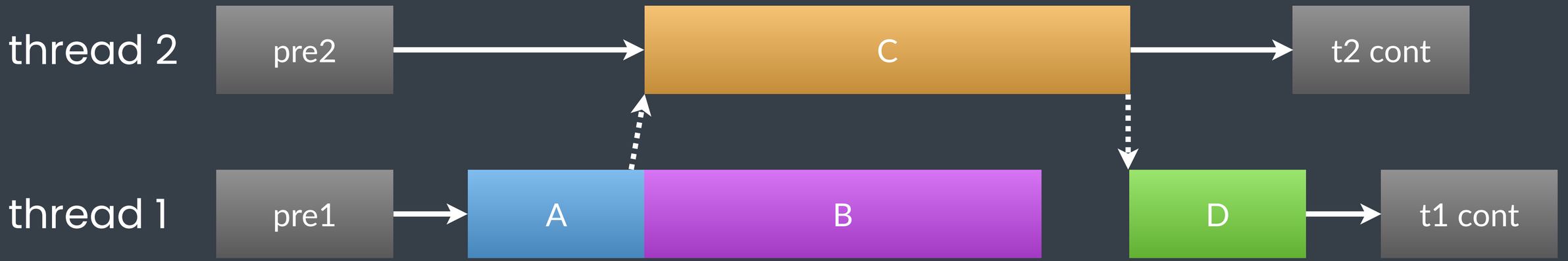
at runtime



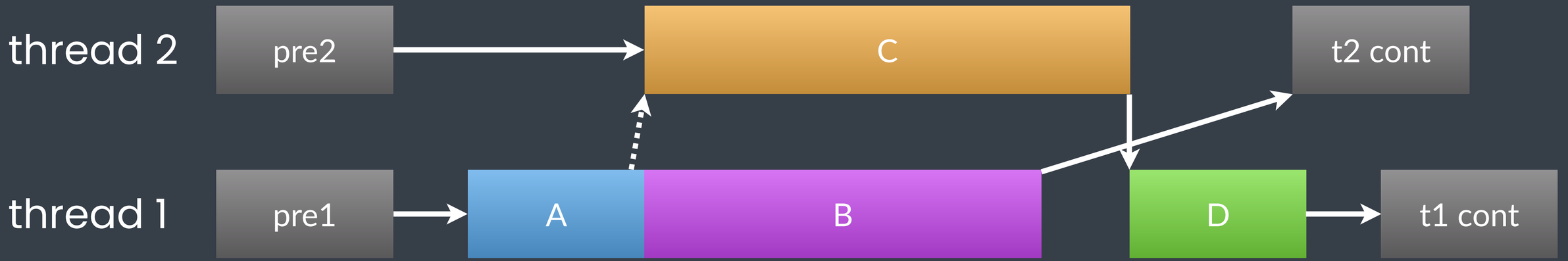
at runtime



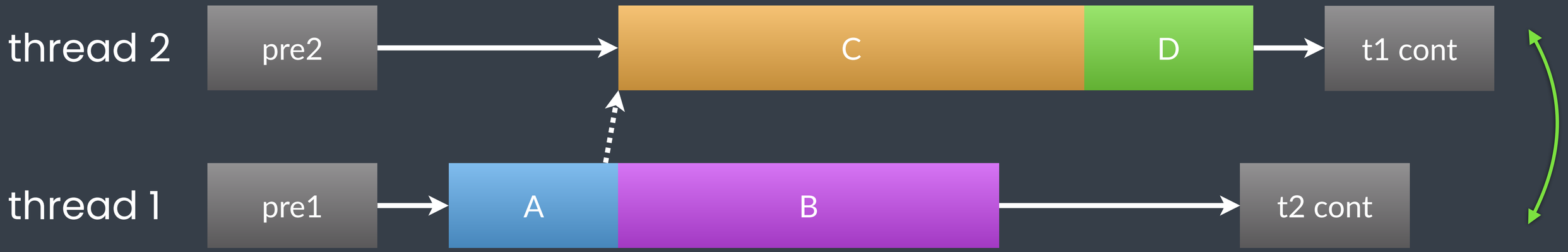
at runtime



at runtime



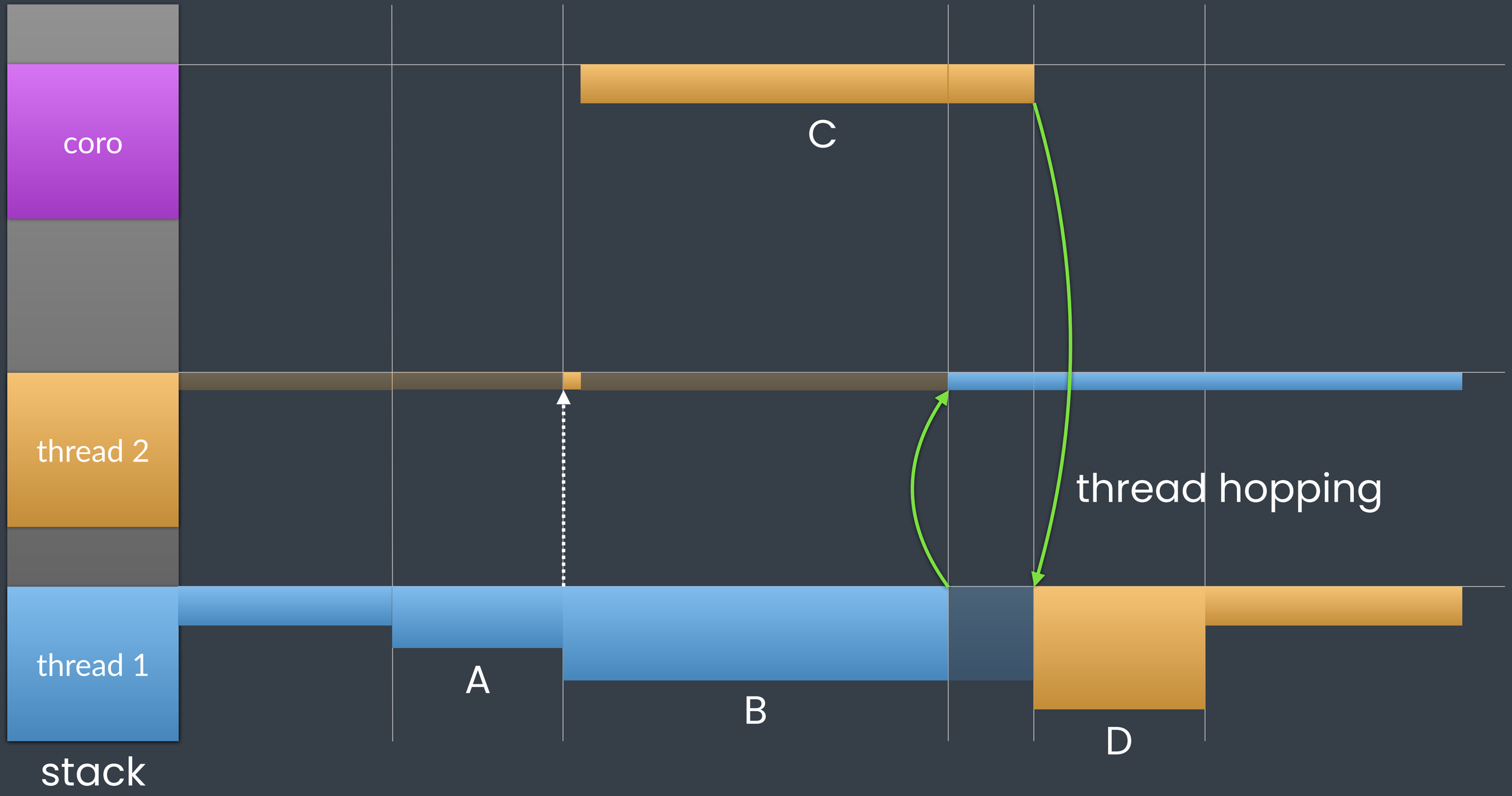
at runtime



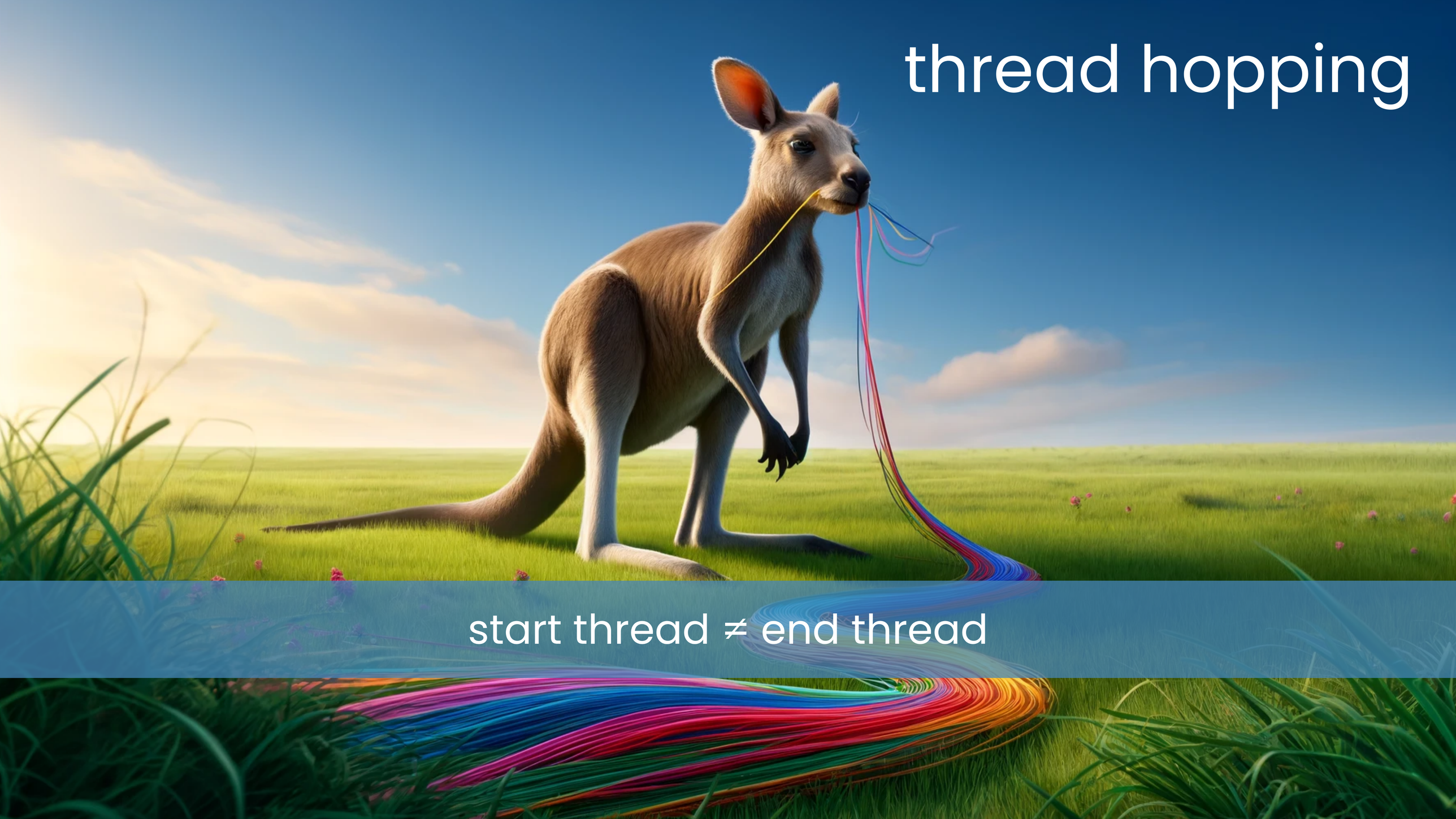
at runtime

stackfull coroutines

using `boost::context`


coro
thread 2
thread 1
stack
C
A
B
D
thread hopping

thread hopping

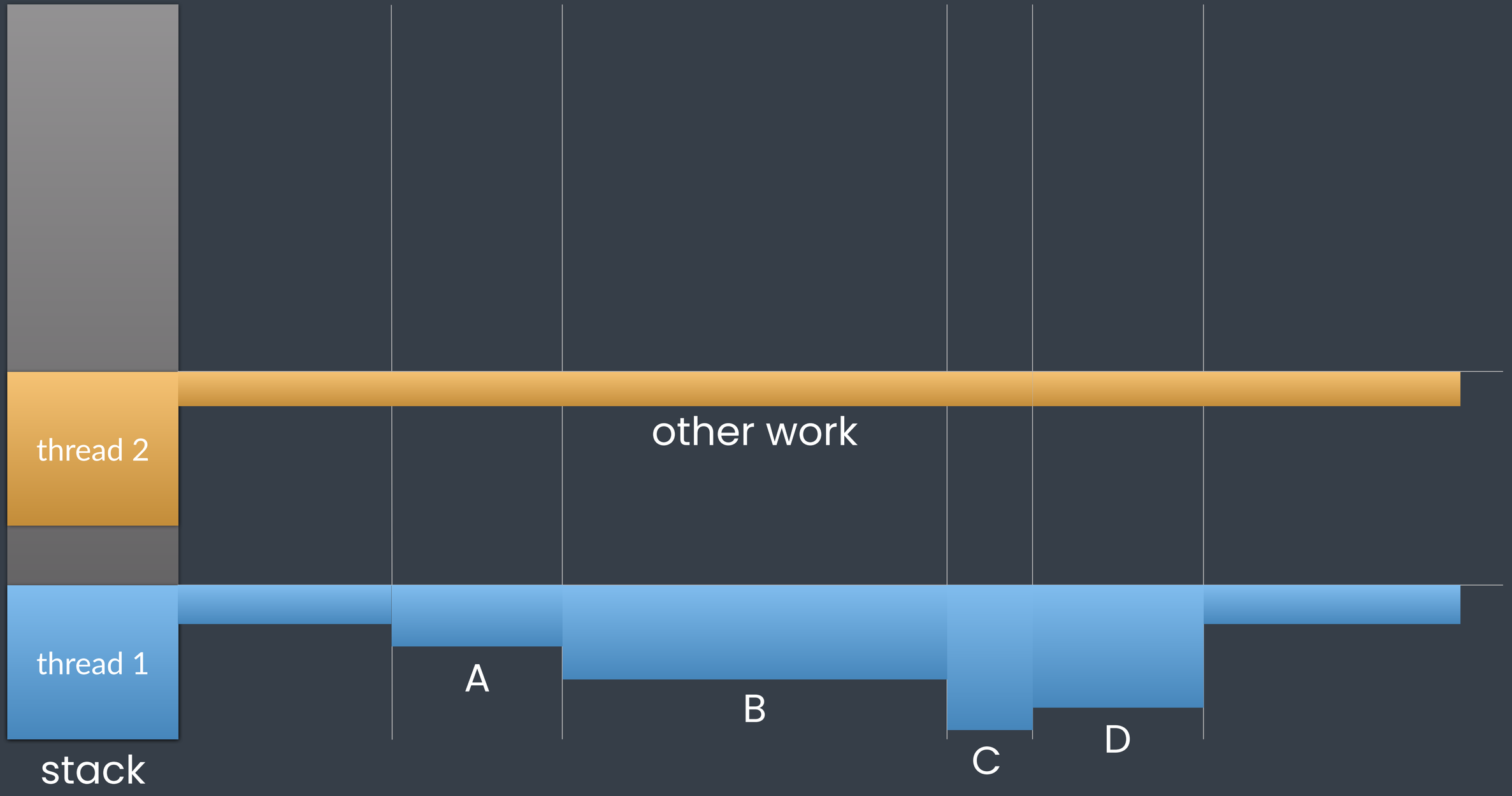


start thread ≠ end thread

2.

no threads when spawning

execute task inside `await`



inplace execution

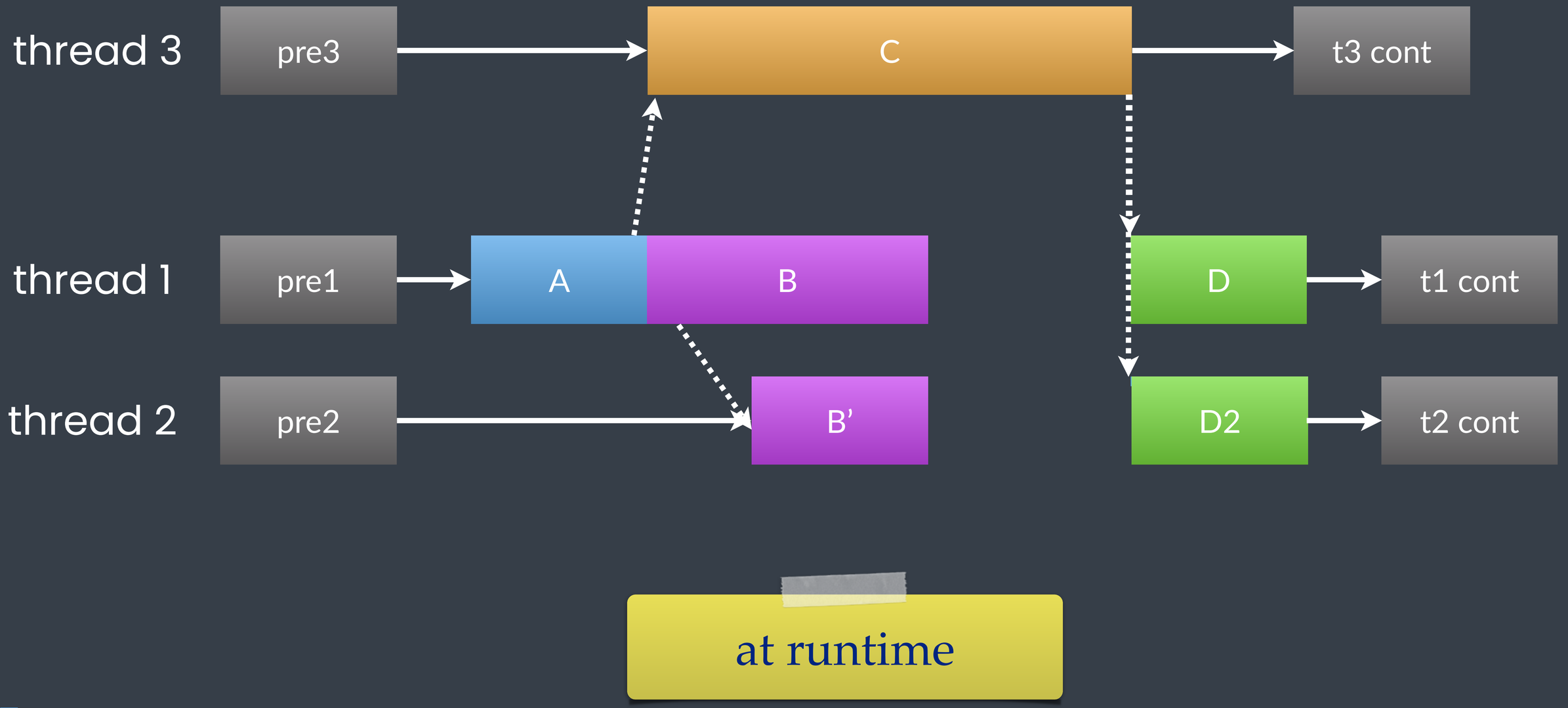
3.

copyable futures

multiple `awaits` waiting on one task

```
A()

var f1 = spawn { C() }
var f2 = f1

// pass f2 to a different thread
B()

…
f1.await()

D()
```

```
…
f2.await()

D2()
```


thread 3
pre3
C
t3 cont
thread 1
pre1
A
B
D
t1 cont
thread 2
pre2
B’
D2
t2 cont

at runtime

suspend

temporary join the work pool
possible thread hopping on resume

Asynchrony

5

synchronous function

```
fun fwrite(_ data: MemoryAddress, _ size: Int, _ count: Int, _ stream: File) -> Int
```

asynchronous function

```
fun fwrite(_ data: MemoryAddress, _ size: Int, _ count: Int, _ stream: File) -> Int
```

asynchrony
is **abstracted away**

asynchrony / concurrency
is **abstracted away**

abstracting concurrency

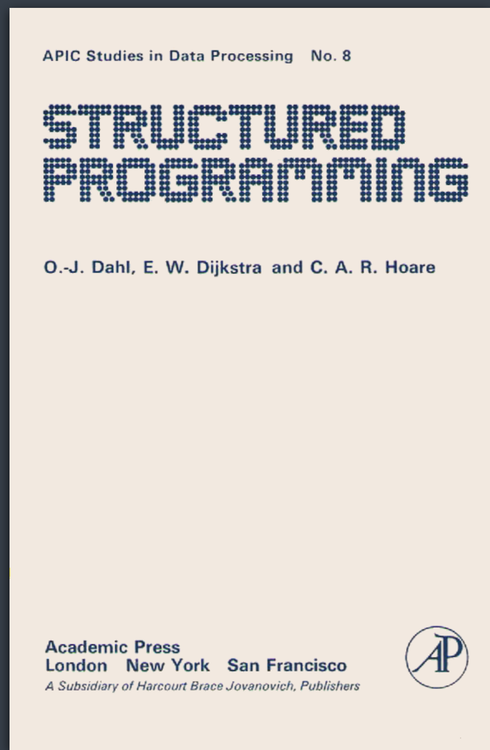
```
fun process_image(_ image: Image) -> ImageResult
```

is it synchronous?

using multiple threads?

1. use of abstractions

abstractions can be used as operations



local reasoning

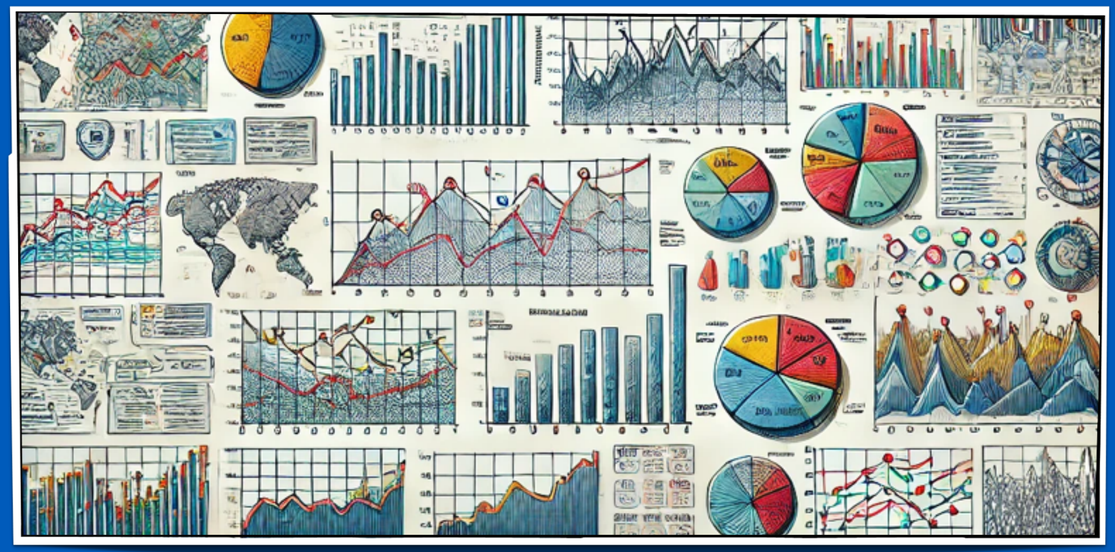
```
fun handle_incoming_connection(_ connection: HttpConnection) {
  do {

    // Read the HTTP request from the socket.
    let request = try get_request(connection)
    request.validate()
    // Handle the request.
    let response = try handle_request(request)
    // Send back the response.
    send_response(response)

  } catch InvalidRequestError(let details) {
    send_response(BadRequestResponse(details), to: connection)
  } catch CancelledError {
    send_response(GatewayTimeoutResponse(), to: connection)
  } catch {
    send_response(InternalServerErrorResponse(), to: connection)
  }
}
```

local reasoning

```
fun get_request(_ connection: inout HttpConnection) throws: HttpRequest {
  // Do the reading on the network I/O scheduler.
  net_scheduler.activate()

  // Read the incoming data in chunks, and parse the request while doing it.
  var parser: HttpRequestParser
  var buffer = MemBuffer(1024*1024)
  while !parser.is_complete {
    let n = connection.read(to: &buffer) // may be an async operation
    &parser.parse_packet(buffer, of_size: n)
  }

  // Done reading; switch now to main scheduler.
  main_scheduler.activate()
  return parser.request
}
```

concurrency is an

implementation detail



Analysis

6

1. modelling concurrency

structured concurrency

easy to express concurrency

2. safety

no race conditions
no additional synchronization
forward progress guarantee
no deadlocks

3. performance

no blocking waits
small `spawn` / `await` synchronisation

4. stack usage

small stack for worker threads

coro ~ # threads => not a lot of stack needed

small stacks for thread hopping

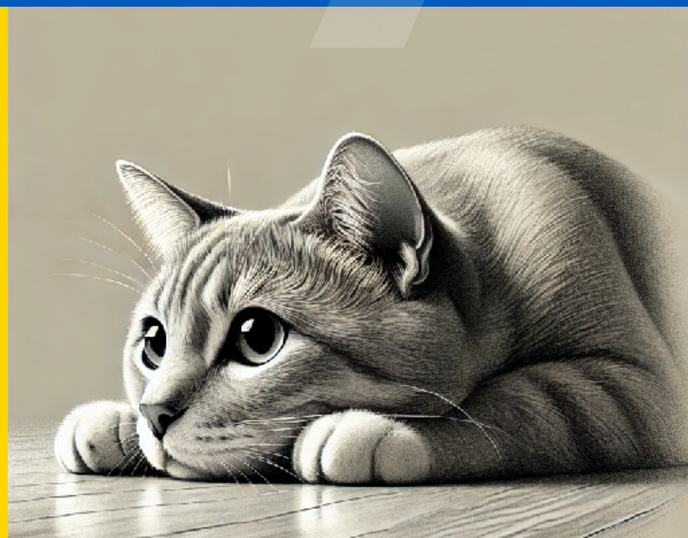
=> stack usage is decent

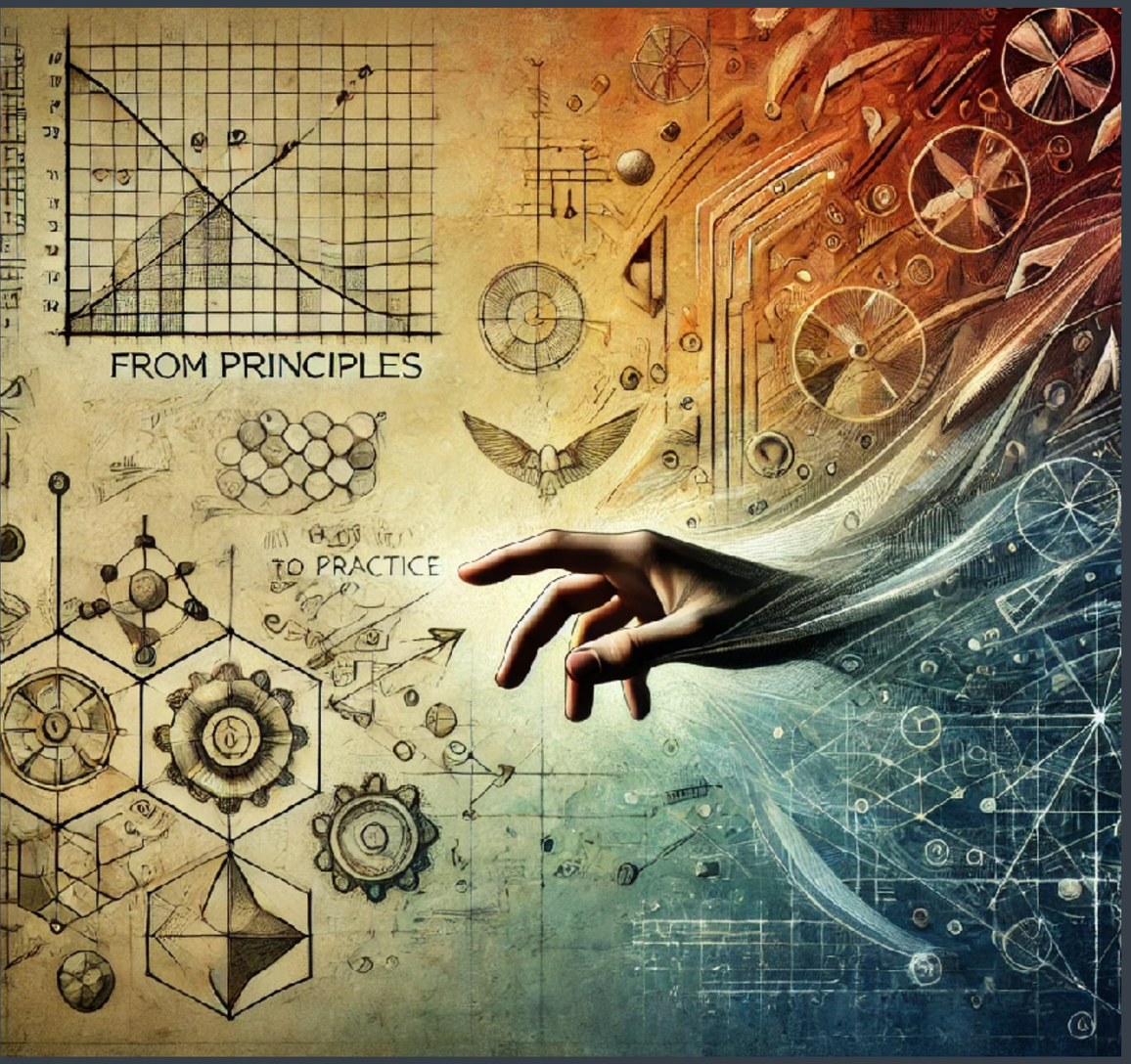
5. interoperability

no thread-local-storage
harder to interop with external modules
(may require sync-wait)

Conclusions

7



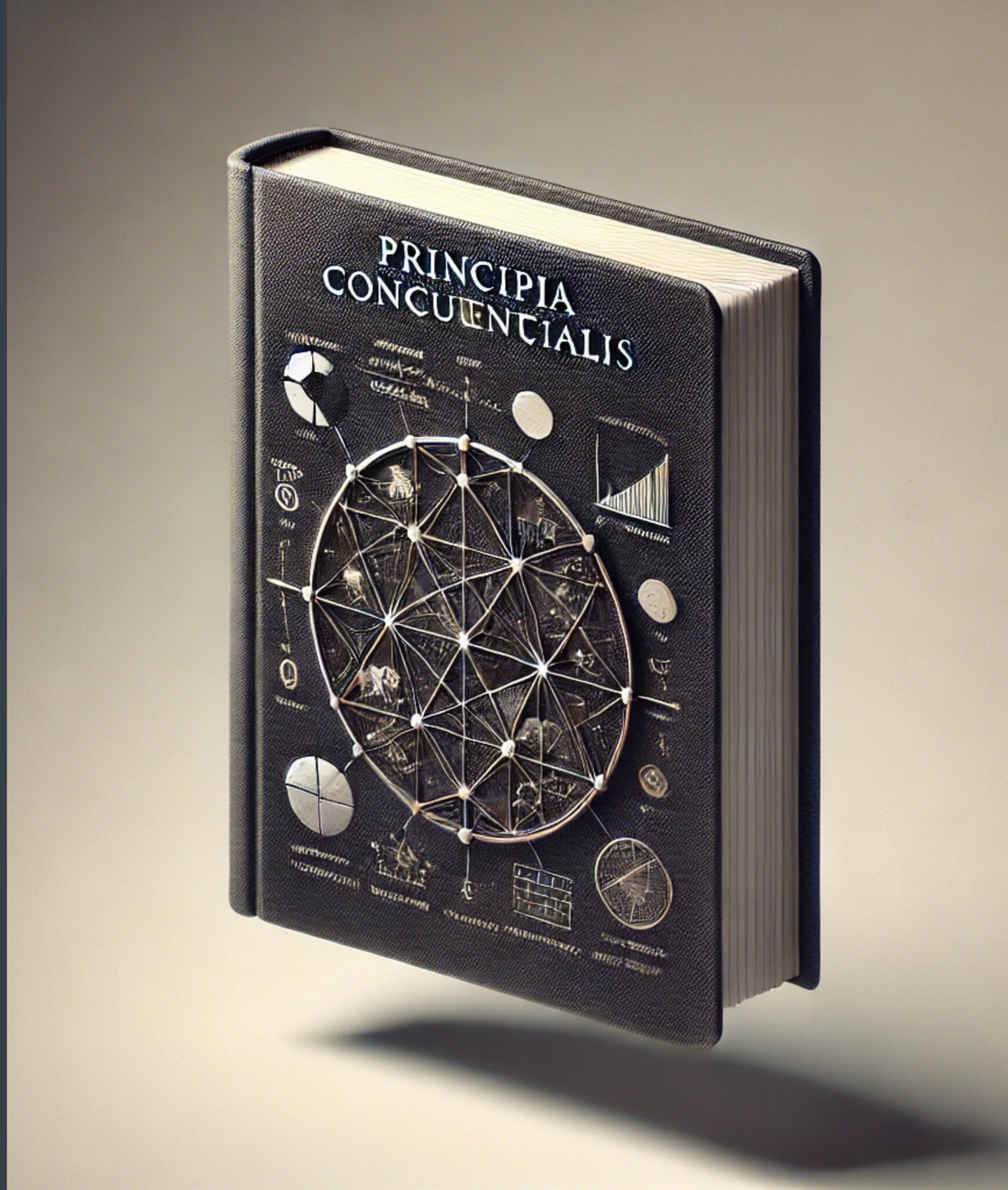


approach

from principles
to practice

principles

structured programming
concurrency
threads & stacks



modeling concurrency

concurrency = expressing constraints
only 3 possibilities at runtime
design time: 4 basic constraints

concurrency in Hylo

easily express concurrency with `spawn` / `await`
no need for a different style
no function colouring
no need for additional synchronisation
structured concurrency

concurrency in Hylo

abstracting concurrency details
local reasoning

reasonable concurrency

additional benefits

safety: no race conditions, no deadlocks
performance: generally fast, scalable, no oversubscription
memory: decent stack consumption

Hylo concurrency

simple syntax / semantics
concurrent code ~ sequential code
concurrency can be abstracted
structured concurrency
local reasoning

Hylo concurrency

simple syntax / semantics

concurrent **code** ~ sequential code

concurrency can be **abstracted**

structured concurrency

local **reasoning**

Hylo concurrency

simple syntax / semantics

concurrent **code** ~ sequential code

concurrency can be **abstracted**

structured concurrency

local **reasoning**

reasonable concurrency

Thank You

@LucTeo@techhub.social
lucteo.ro