

Core C++ 2026

20-21Oct. 2026 :: Tel-Aviv

From Silicon to C++26

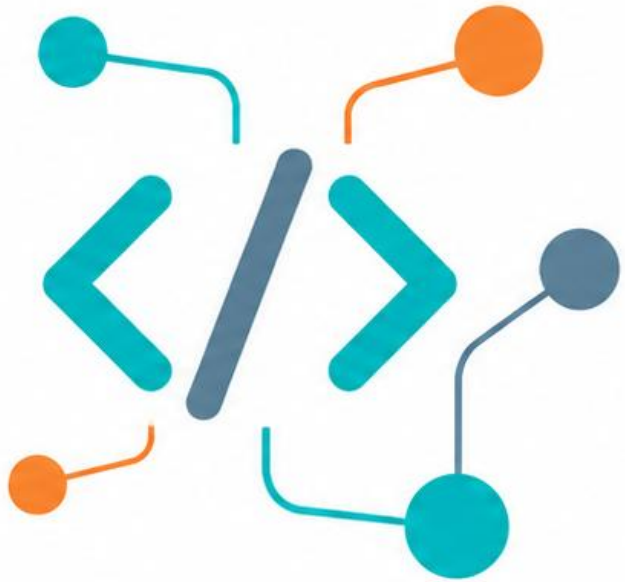
Miri (Kopel) Ben-Nissan, Ph.D

Miri (Kopel) Ben-Nissan

Ph.D. in Computer Science

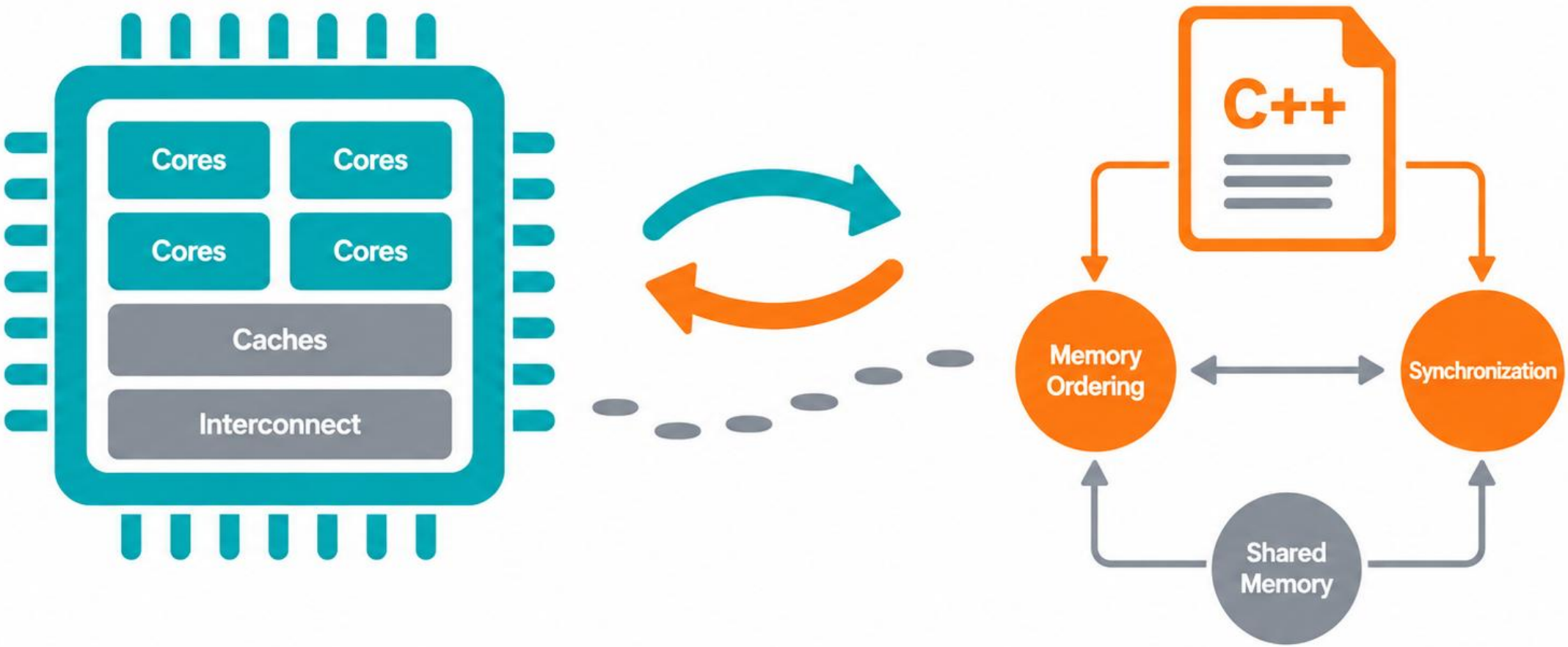
C C++ Python

miri@mkbn.co.il

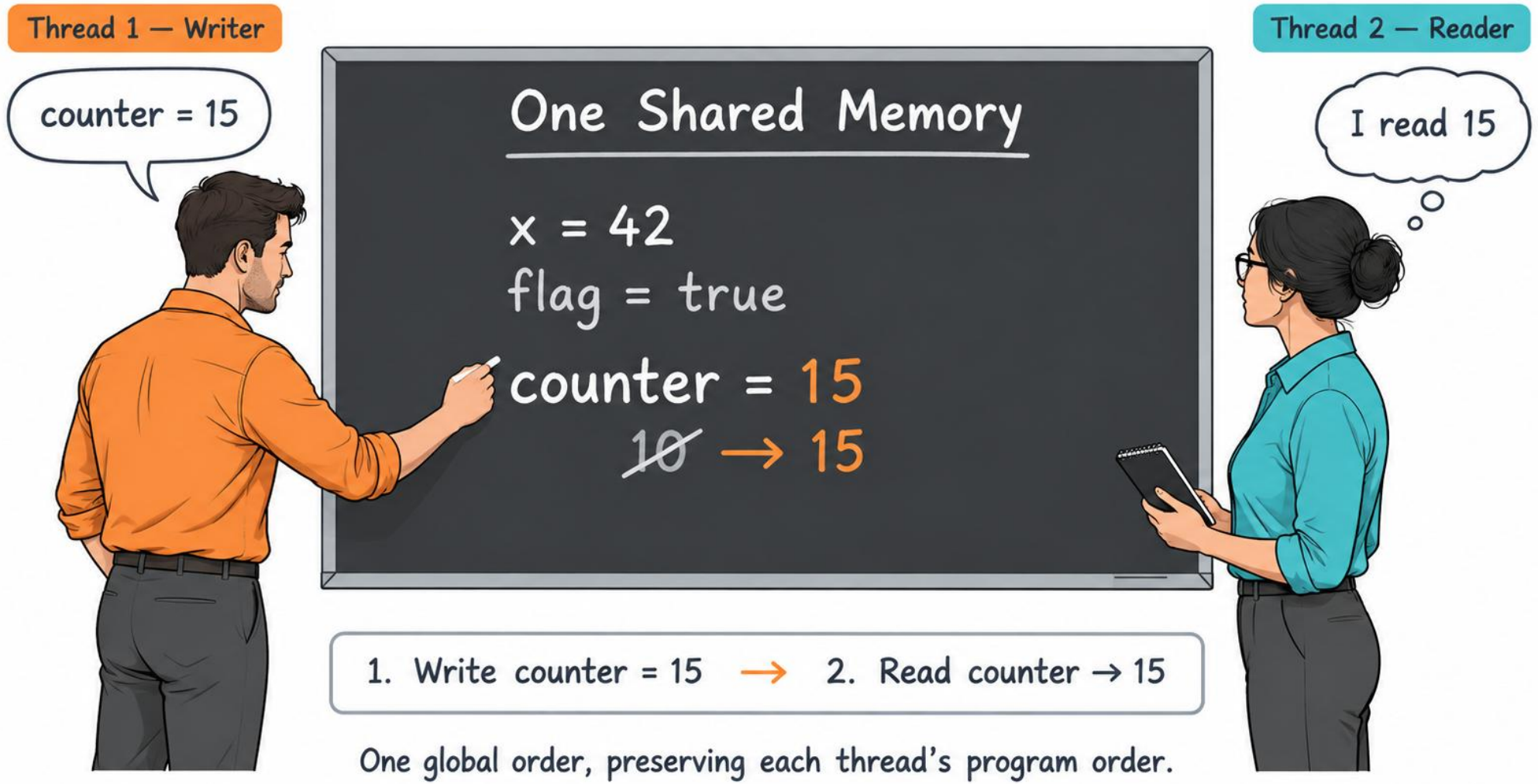


Hardware Meets Software

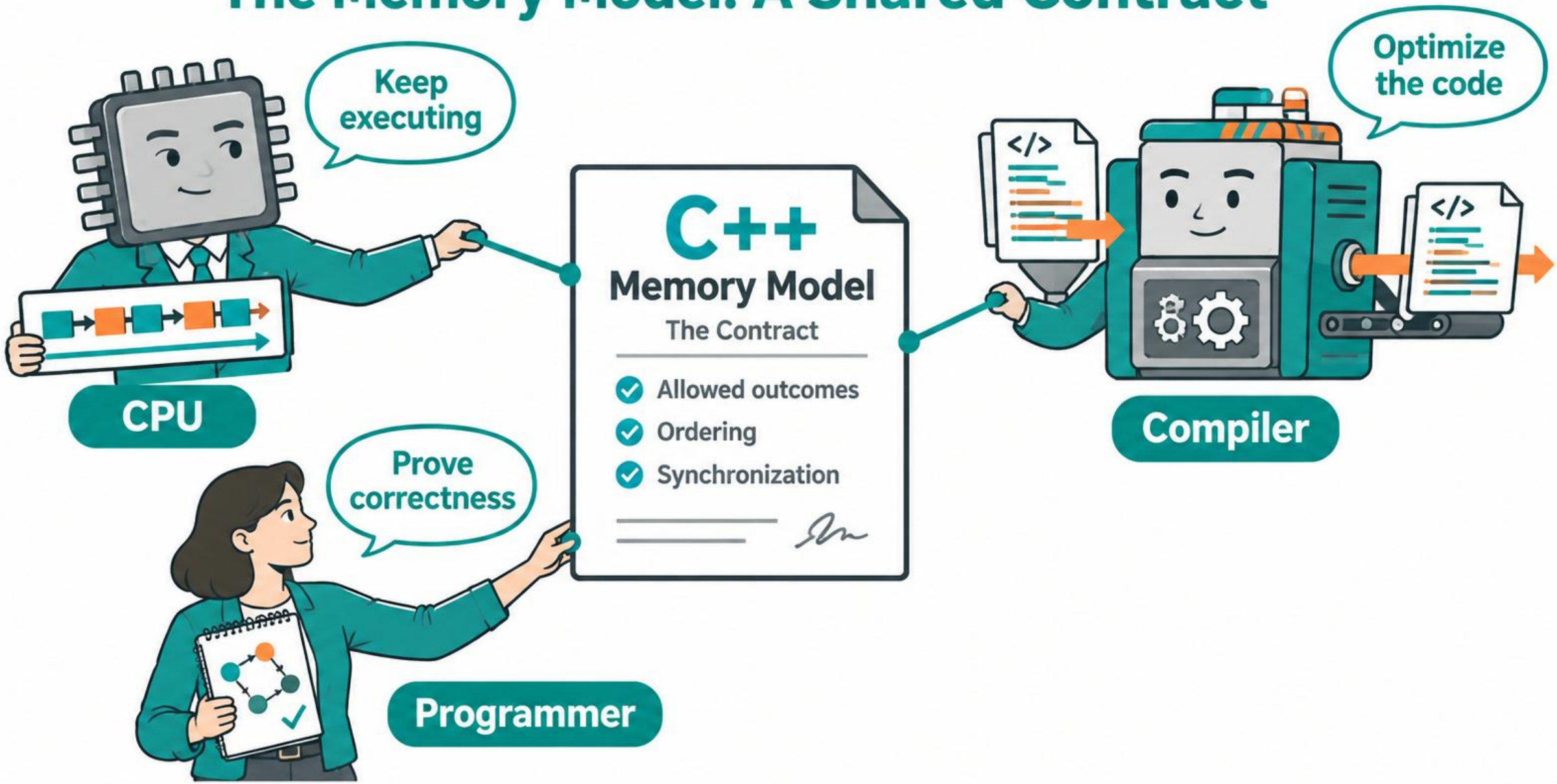
A journey through processor architecture and the evolution of the C++ memory model



Introduction: The Illustration of Linear Memory



The Memory Model: A Shared Contract



Freedom to execute and optimize — within guarantees we can reason about.

Four Layers We Must Keep Separate

1. C++ Program / Language Semantics



Types, evaluations, object access, lifetime, atomics, ordering

Defines data races and allowed outcomes

↓ Compile

2. Compiler

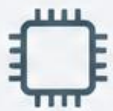


Instruction selection, optimization, scheduling, register allocation

Must preserve the semantics of well-defined programs

↓ Execute

3. ISA / Architectural Memory Model



Observable behavior, ordering guarantees, instruction semantics

Examples across ISAs: MOV • LDAR • STLR • DMB

↓ Implement

4. Microarchitecture



Store buffers, queues, speculation, caches, interconnect, coherence

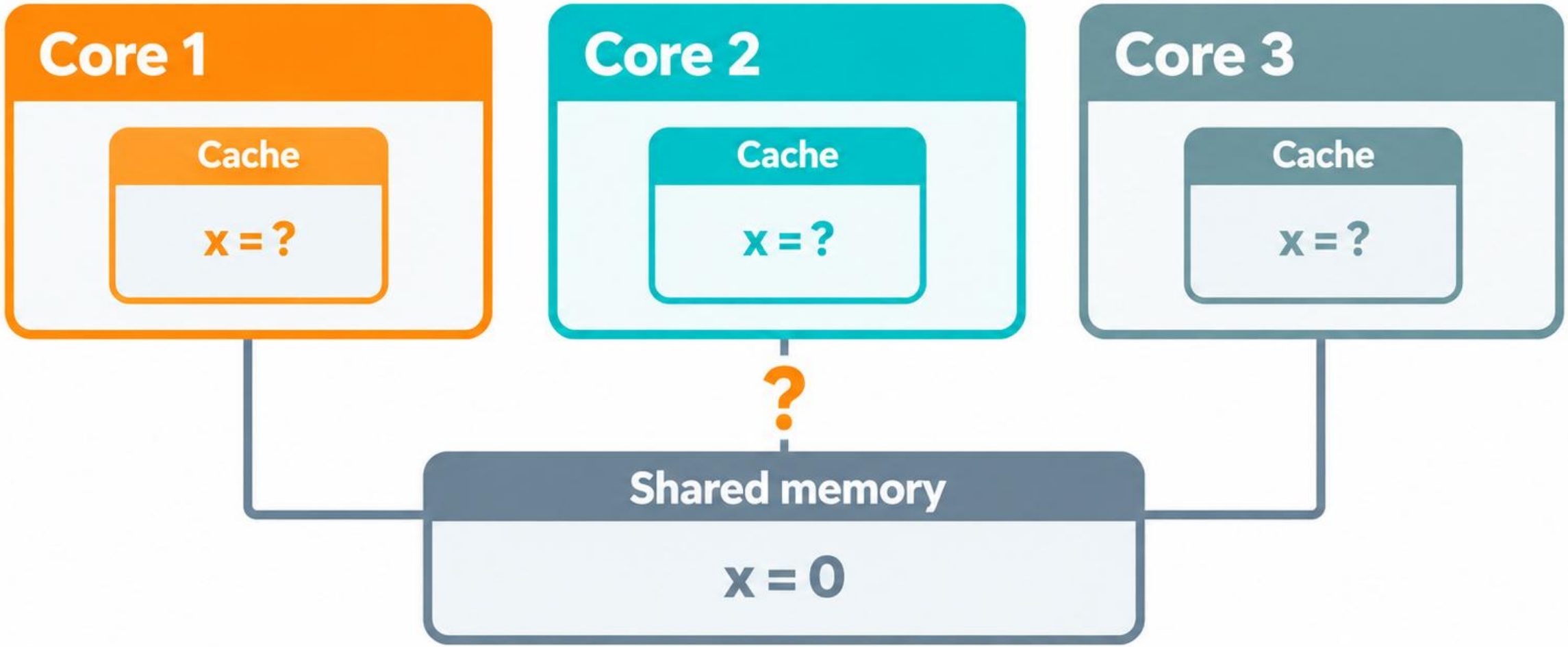
Different implementations must satisfy the same architectural contract

release → implementation that satisfies the publication guarantee

Not a universal command to flush the entire store buffer

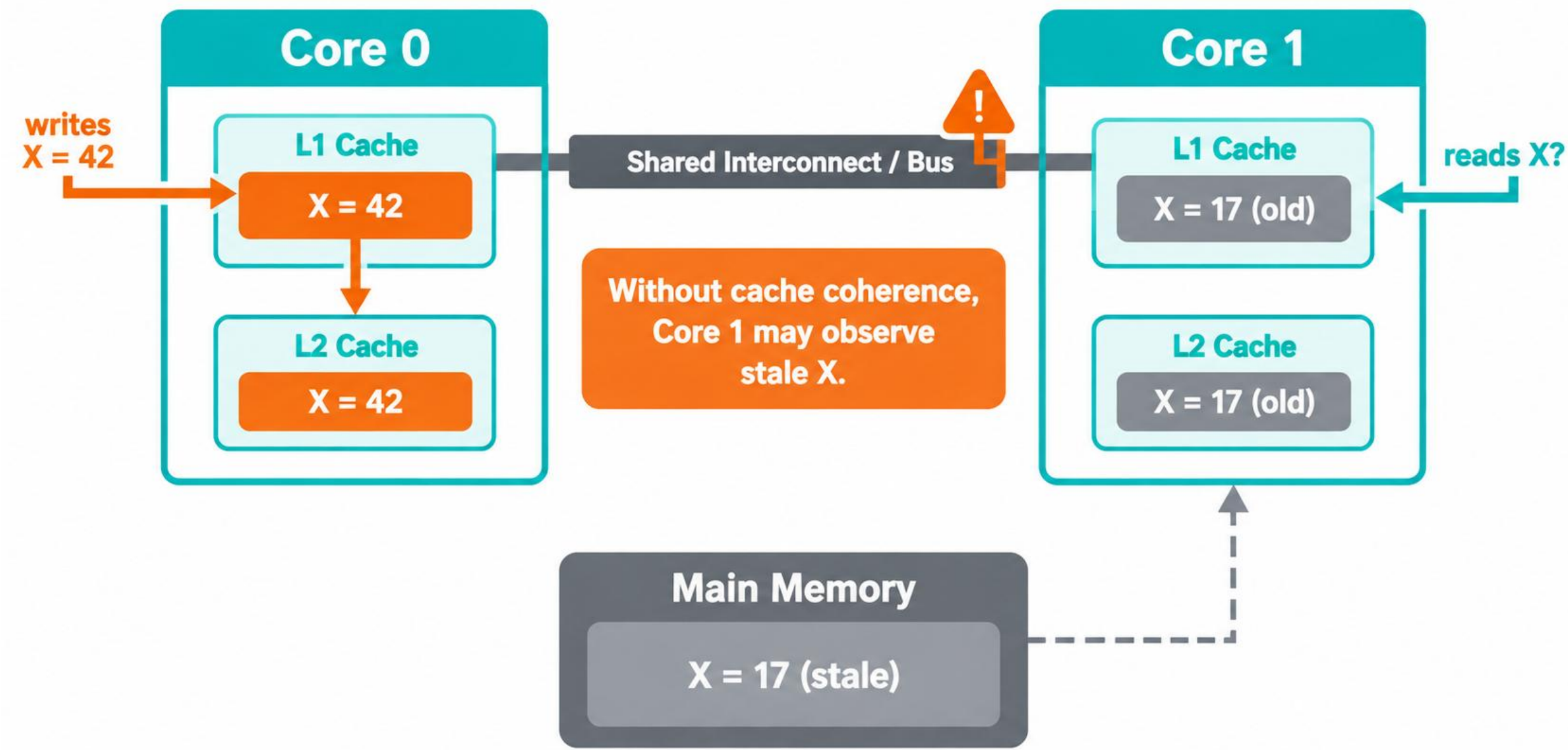
How do cores keep a shared view of memory?

Multiple cores. Each with its own cache.



Who sees what — and when?

Why do we need a cache coherency protocol



MESI Cache-Line States

M

Modified

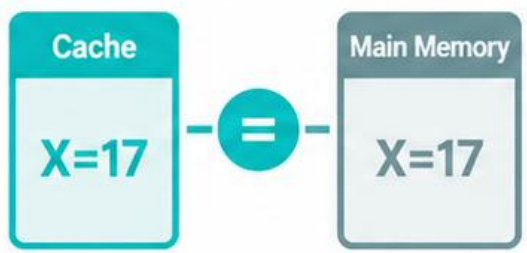
Only this cache holds the line. Data is dirty — different from main memory.



E

Exclusive

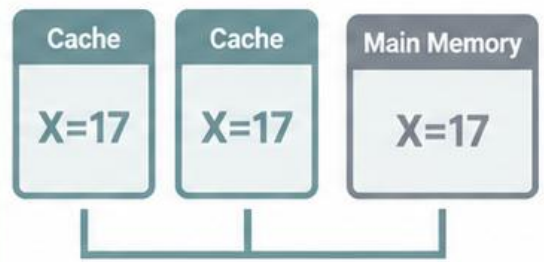
Only this cache holds the line. Data is clean — matches main memory.



S

Shared

The line is valid in one or more caches. All copies match main memory.



I

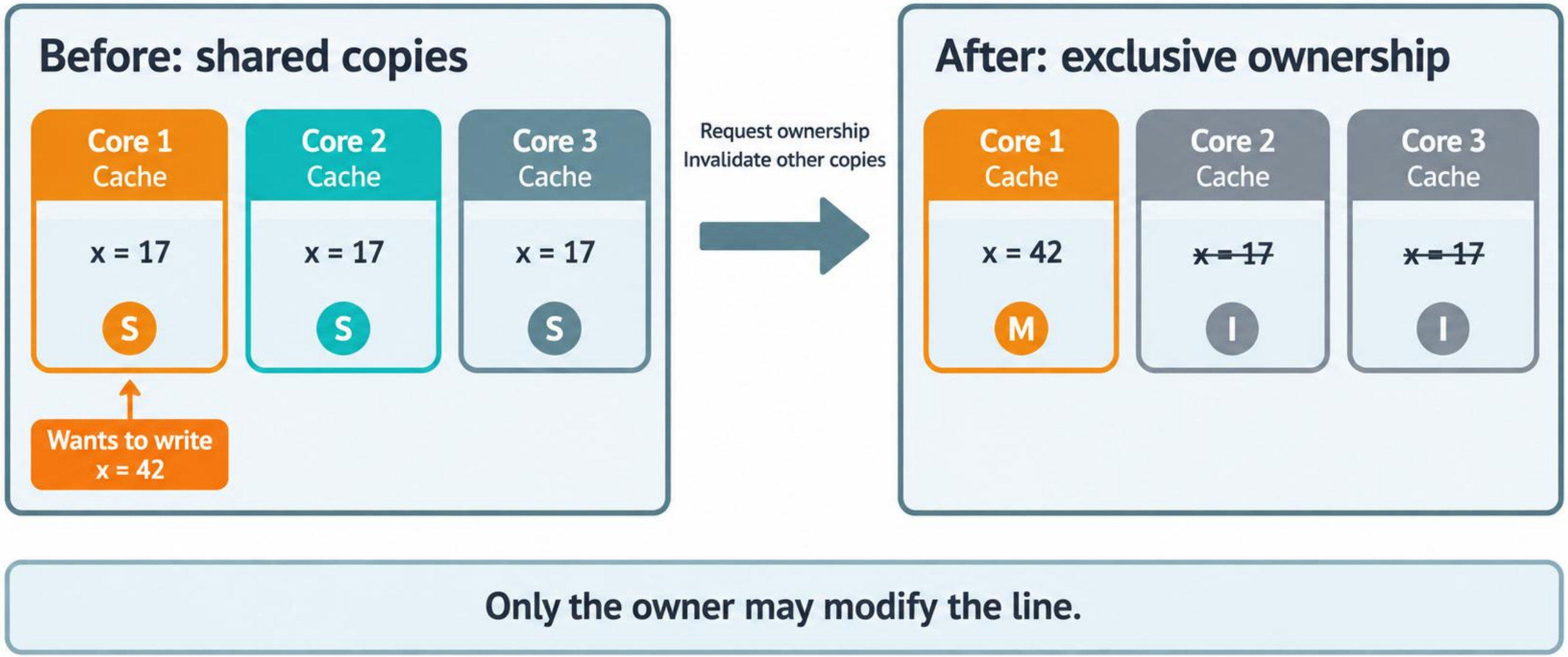
Invalid

No valid data is held.
The cache line cannot be used.



Writing requires exclusive ownership

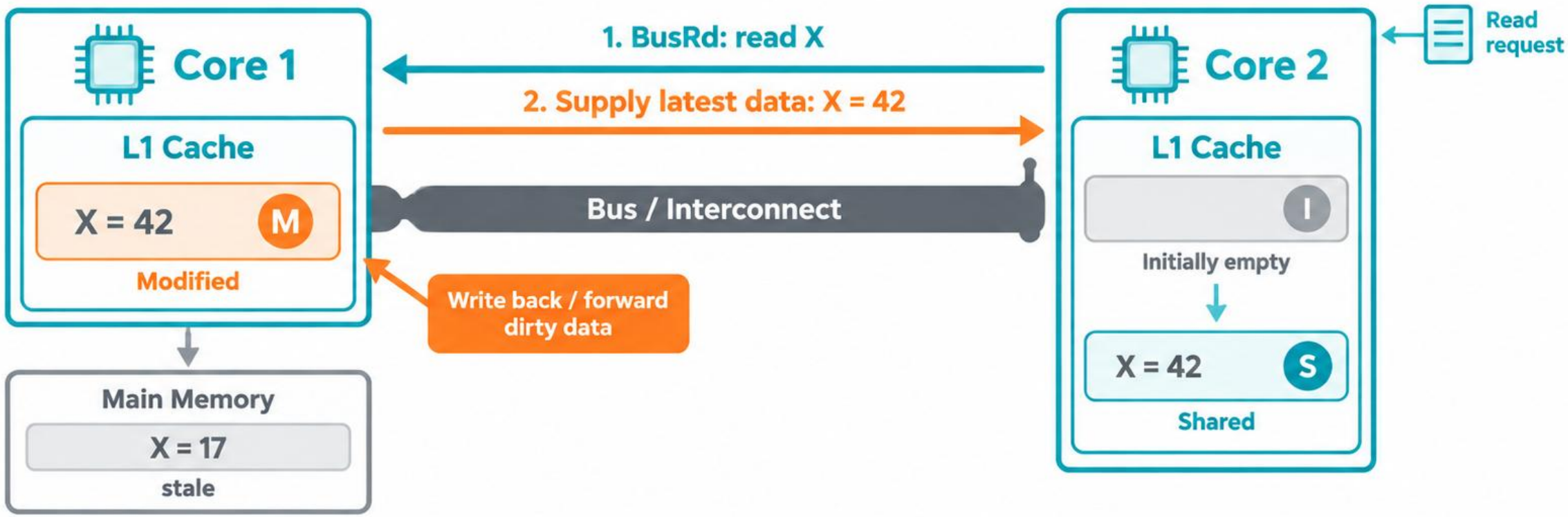
Before modifying a shared cache line, a core must obtain write permission and invalidate other copies.



Only the owner may modify the line.

MESI Transition: Modified → Shared

Core 2 reads a line dirty in Core 1



After coherence transaction

Core 1 L1	Core 2 L1	Main Memory
X = 42 S	X = 42 S	X = 42
Shared	Shared	updated

What if every store had to wait?

A simplified model without a store buffer

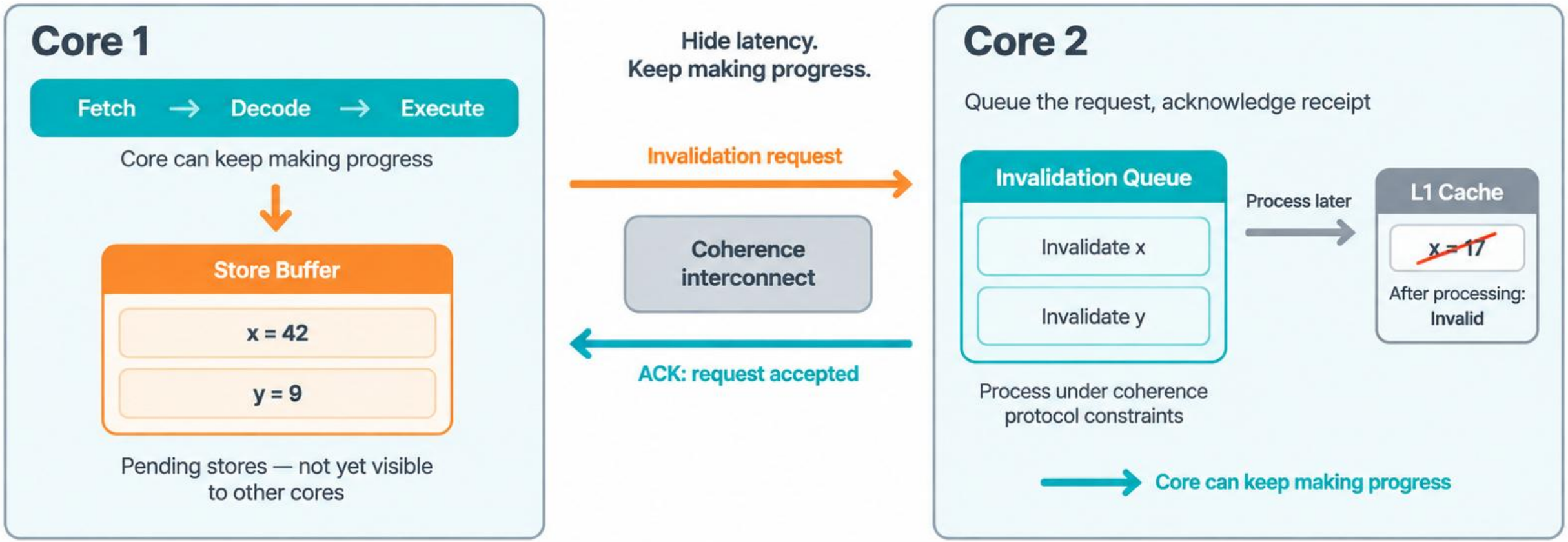


Request ownership → wait for coherence → update the cache line

Can the core keep working while ownership is pending?

Keeping the CPU Pipeline Moving

A simplified hardware model



Store buffers hide write latency
Invalidation queues absorb coherence traffic

```cpp
std::atomic<int> x{0};
std::atomic<int> y{0};

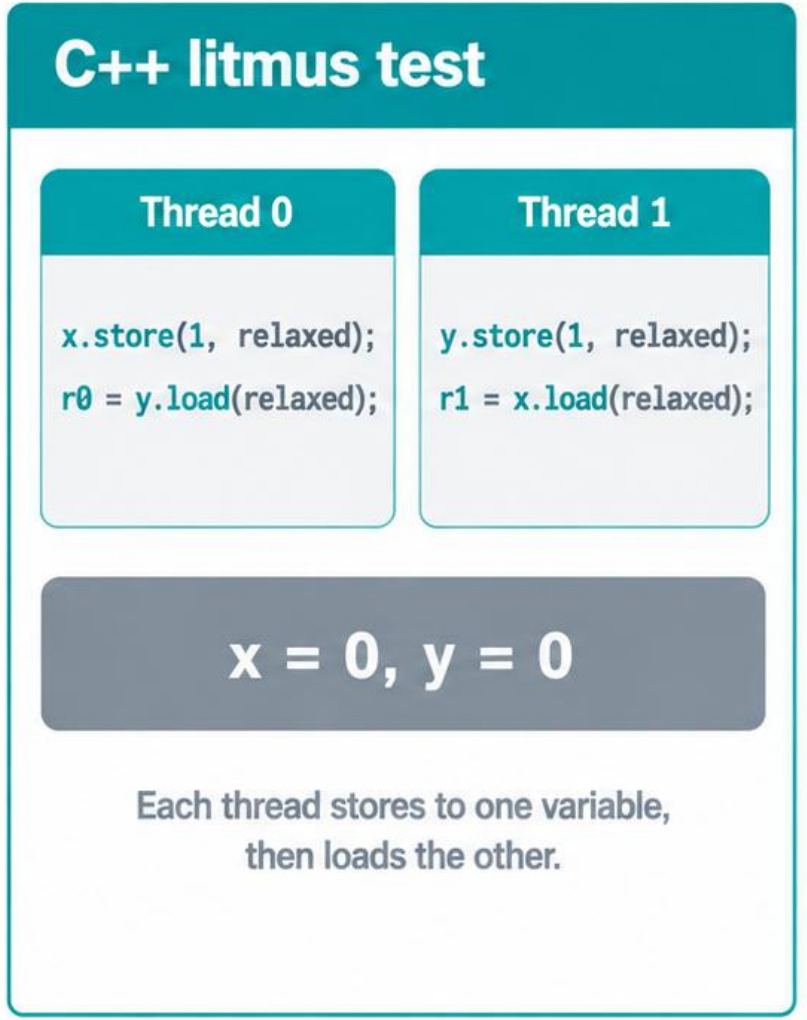
std::barrier sync_point(3);

int r0 = 0;
int r1 = 0;

int main()
{
    std::thread t0([] {
        for (std::uint64_t i = 0; i < rounds; ++i) {
            sync_point.arrive_and_wait();

            x.store(1, std::memory_order_relaxed);
            r0 = y.load(std::memory_order_relaxed);

            sync_point.arrive_and_wait();
        }
    });
```



C++ litmus test

Thread 0	Thread 1
`x.store(1, relaxed);`	`y.store(1, relaxed);`
`r0 = y.load(relaxed);`	`r1 = x.load(relaxed);`

x = 0, y = 0

Each thread stores to one variable, then loads the other.

```
std::thread t1([] {
    for (std::uint64_t i = 0; i < rounds; ++i) {
        sync_point.arrive_and_wait();

        y.store(1, std::memory_order_relaxed);
        r1 = x.load(std::memory_order_relaxed);

        sync_point.arrive_and_wait();
    }
});
```

```
    for (std::uint64_t i = 0; i < rounds; ++i) {
        x.store(0, std::memory_order_relaxed);
        y.store(0, std::memory_order_relaxed);

        sync_point.arrive_and_wait(); // release both workers
        sync_point.arrive_and_wait(); // wait for both workers to finish

        out << r0 << " | " << r1 << '\n';
    }

    t0.join();
    t1.join();
}
```

Three Outcomes from Different Interleavings

Initially x = 0, y = 0 • Result = (r0, r1)

(1,1)

Time	Thread 0	Thread 1
1	x = 1	
2		y = 1
3	r0 = y → 1	
4		r1 = x → 1

(1,0)

Time	Thread 0	Thread 1
1		y = 1
2		r1 = x → 0
3	x = 1	
4	r0 = y → 1	

(0,1)

Time	Thread 0	Thread 1
1	x = 1	
2	r0 = y → 0	
3		y = 1
4		r1 = x → 1

Each thread preserves its order: store first, then load.

Simplified model: each write is immediately observable.

DEMO

C++ litmus test

Thread 0	Thread 1
`x.store(1, relaxed);`	`y.store(1, relaxed);`
`r0 = y.load(relaxed);`	`r1 = x.load(relaxed);`

x = 0, y = 0

Each thread stores to one variable, then loads the other.

Possible outcomes (r0, r1)

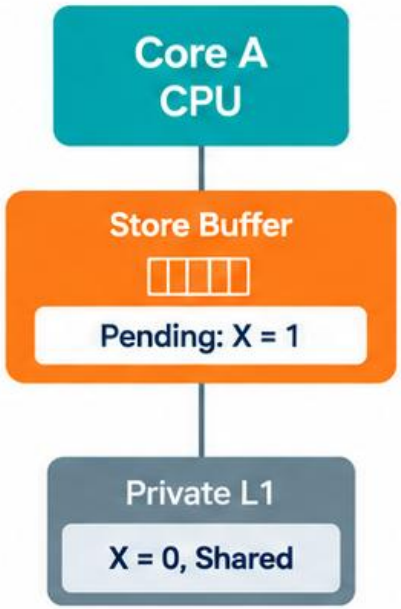
- (1,1)
- (1,0)
- (0,1)
- (0,0) — POSSIBLE

With relaxed atomics; 0,0 is allowed, not guaranteed.

When Does a Store Become Visible?

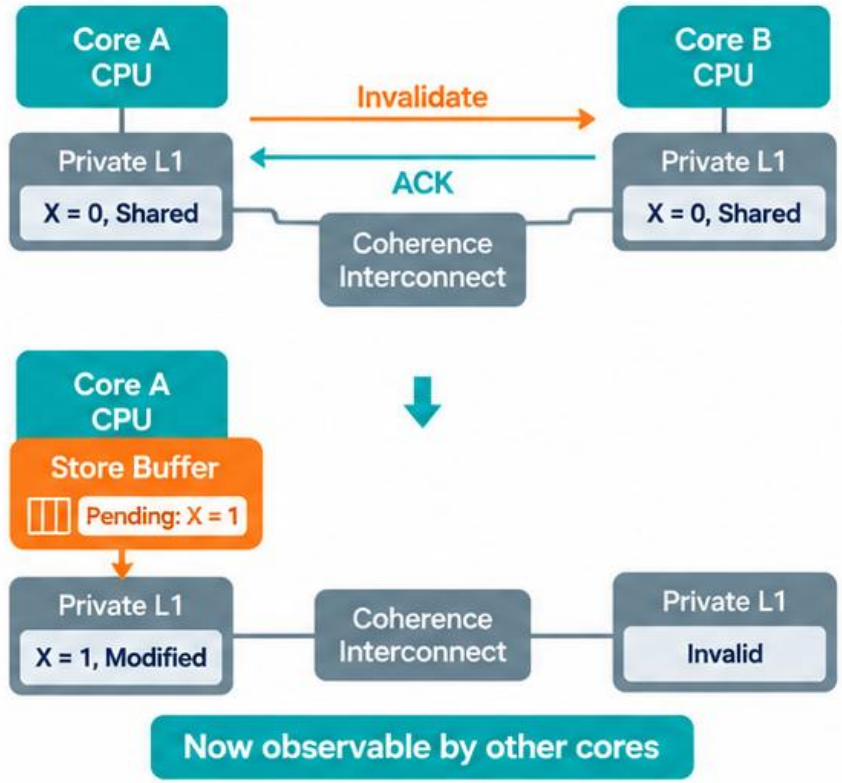
Visibility through coherent caches — not necessarily DRAM

1. Buffer the store



Execution can continue while the store is pending.

2. Acquire ownership & commit

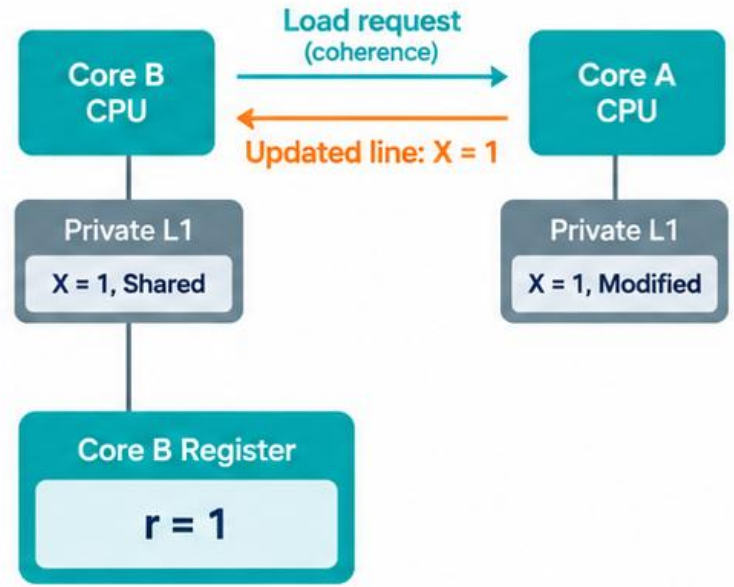


Now observable by other cores

Invalidate sharers if needed, then commit to coherent cache.

If already E/M, no new invalidation is needed.

3. Another core observes it

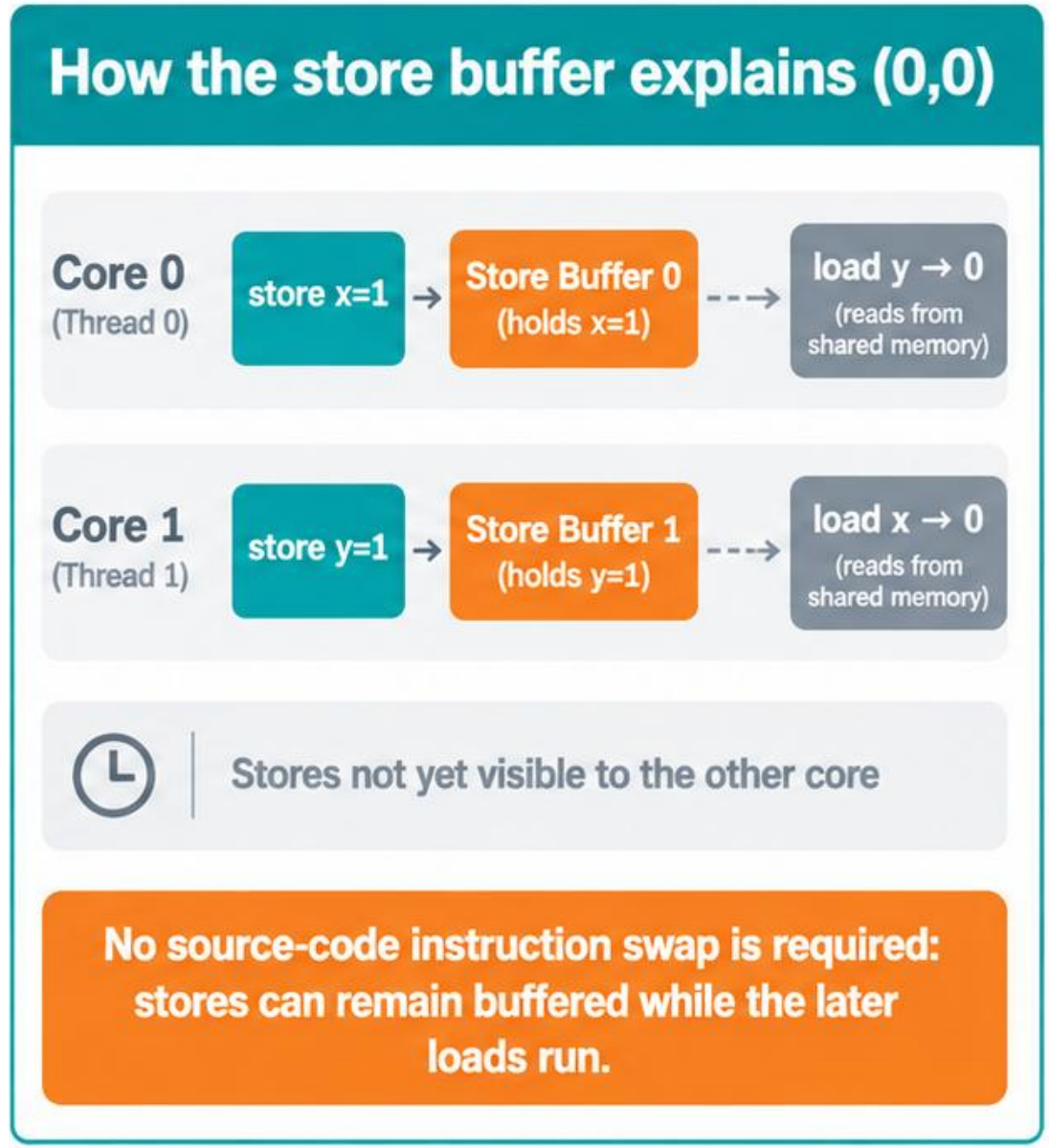


A later load obtains the updated value via coherence.

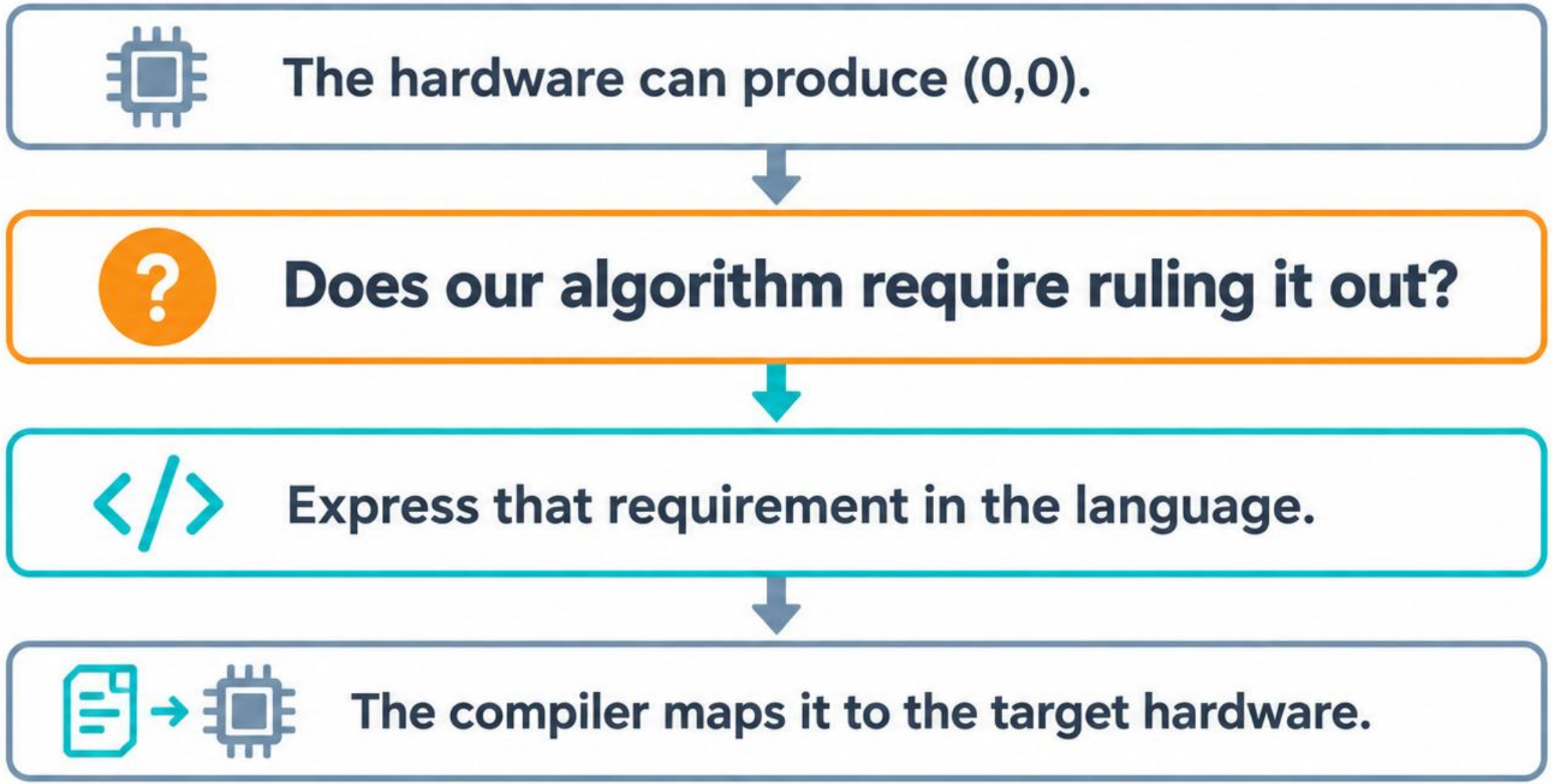
State observation is distinct from becoming observable.

Simplified hardware model; details vary by CPU. Fences enforce ordering; release is not a universal flush.

Store Buffering: Why Can We Get (0,0)?

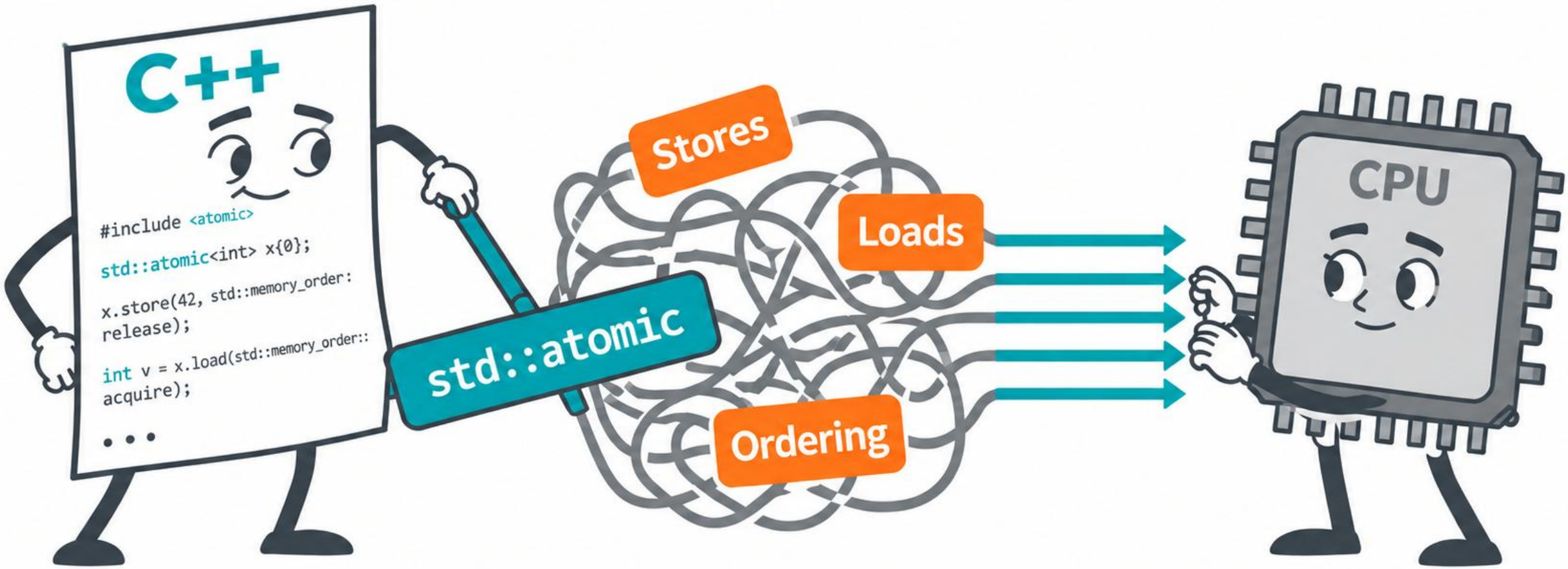


From Hardware Behavior to a Language Contract



Specify the guarantee — not the implementation mechanism.

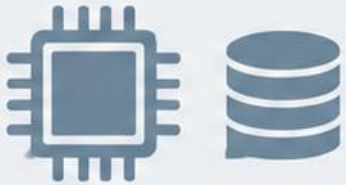
Code Comes to the Rescue



Software expresses the requirements.
Compiler and hardware enforce them.

From Hardware Mechanisms to Language Guarantees

Hardware mechanisms



Caches

Store buffers

Coherence traffic

Raise the abstraction

Language guarantees

</>

What can the program rely on?

What may the compiler change?

What must the implementation guarantee?

From how the machine works to what the program can assume.

The memory model is the contract.

Before C++11: the tools

Threading libraries

pthreads / Win32 threads

Compiler support

Intrinsics / compiler barriers

Hardware guarantees

Atomic instructions / memory barriers

Threads existed.

ISO C++98/03 did not define their shared-memory semantics.

The missing language contract

Cross-thread visibility

When does another thread see a write?

Ordering guarantees

Which observations and reorderings are allowed?

Portable low-level synchronization

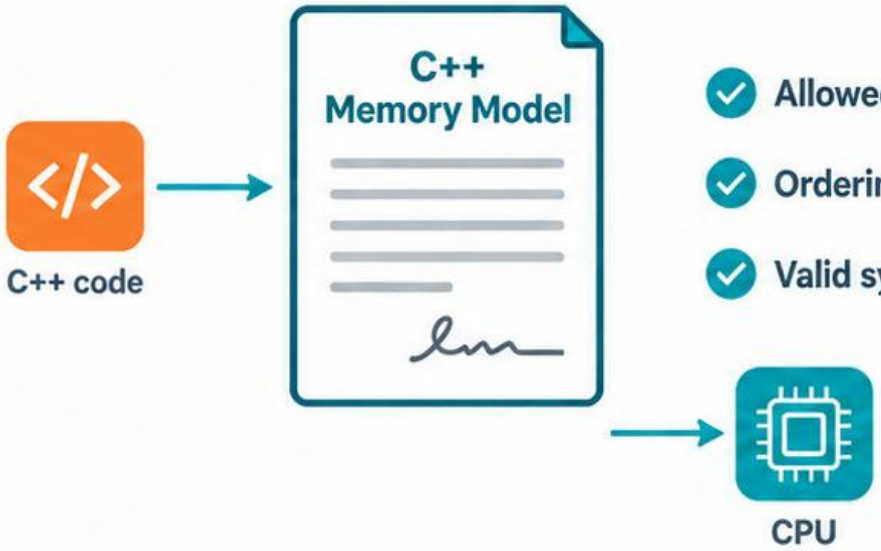
No standard atomics or memory-order controls.

Correctness depended on the library + compiler + hardware contract.

Source: Boehm, PLDI 2005 | C++ memory model history

The C++ Memory Model: A Contract, Not a Schedule

What the contract defines

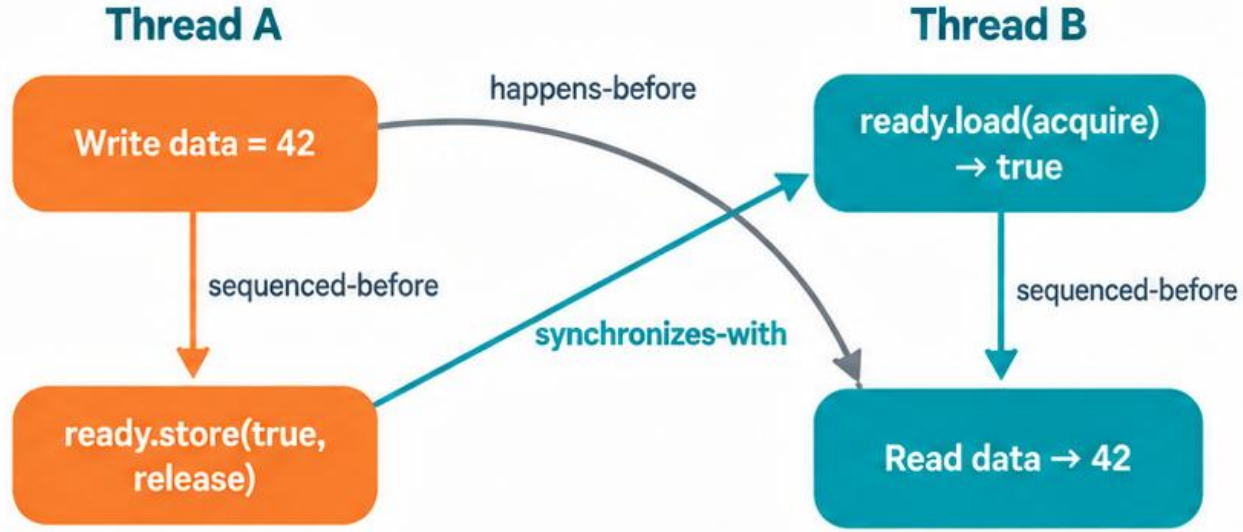


- Allowed outcomes
- Ordering relationships
- Valid synchronization



Not a cycle-by-cycle execution timeline

Reason with a graph



The acquire reads the value written by the release.

Initially ready = false; one publication; no later writes to data.

Nodes = memory operations • Arrows = ordering & synchronization

---

Use the graph to reason about happens-before, data races, and allowed read values.

Expressing Synchronization in C++

Release–Acquire safely publishes data between threads

Initially: ready = false • One publication, no later writes to data

Writer — Core A

```
data = 42;
ready.store(true, std::memory_order_release);
```

Release orders earlier memory operations before publication.

Acquire reads the value written by Release

Reader — Core B

```
if (ready.load(std::memory_order_acquire)) {
    use(data); // sees 42
}
```

Acquire orders subsequent memory operations after observation.

Write data → **Release** → **Acquire** → **Read data**

happens-before

Ordering guarantees — not a universal pipeline stop or queue flush.

- Compiler and CPU enforce the required ordering; implementation depends on architecture.
- Release/acquire alone does not forbid (0,0) in the two-variable Store Buffering test.
- Typical x86 atomic loads/stores need no extra fence for release/acquire.

The Contract Works Both Ways

Publish the data. Acknowledge the read. Reuse safely.

```
int data = 0;
std::atomic<unsigned> published{0}, acknowledged{0};
```

Inside the per-version loops

Writer

```
while (acknowledged.load(std::memory_order_acquire)
       != version - 1) {}

data = (version == 1) ? 42 : 99;

published.store(version, std::memory_order_release);
```

Reader

```
while (published.load(std::memory_order_acquire)
       != version) {}

use(data);

acknowledged.store(version, std::memory_order_release);
```

→ **Publication: write data → release → acquire → read data**

← **Acknowledgment: read data → release → acquire → write next version**

Both directions establish happens-before for the non-atomic data.

The acquire must read the matching release. use(data) completes before acknowledgment.

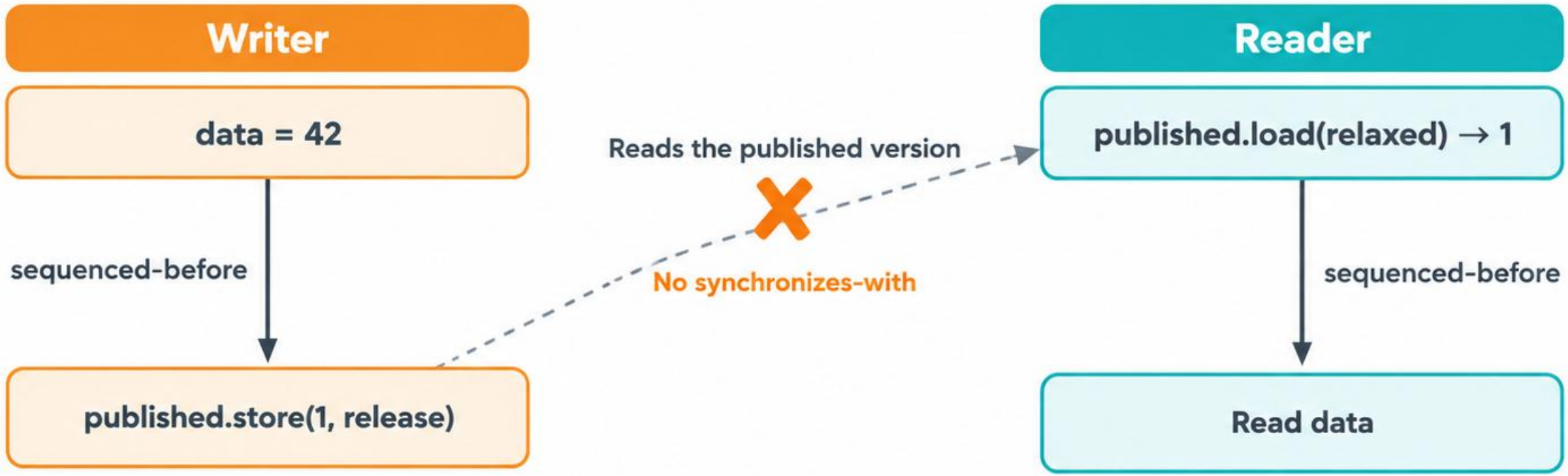
example: Weaken Only the Publication Load

```
// writer(): unchanged
while (acknowledged.load(std::memory_order_acquire)!= version - 1) {
    std::this_thread::yield();
}
data = (version == 1) ? 42 : 99;
published.store(version, std::memory_order_release);
```

```
// reader(): change this load only
while (published.load(std::memory_order_relaxed) != version) {
    std::this_thread::yield();
}
std::cout << "Version " << version
          << ": data = " << data << '\n';
acknowledged.store(version, std::memory_order_release);
```

Seeing the Version Is Not Acquiring the Data

Experiment 1: only the publication load is relaxed



No happens-before from the write to the read of data

Conflicting non-atomic accesses → data race → undefined behavior

The acknowledgment remains release/acquire: it protects reuse, not this publication.

memory_order_consume: Ordering Through Dependencies

C++11: Can a dependency provide the ordering we need?

```
std::atomic<Node*> published{nullptr};
```

One publication; the object remains alive and unchanged.

Writer

```
auto* p = new Node{42};
published.store(p, std::memory_order_release);
```

Pointer publication

Reader — original C++11 intent

```
auto* p = published.load(
    std::memory_order_consume);

if (p)
    use(p->value);
```

Address dependency

The opportunity

On Arm, preserved address dependencies can order these loads without an extra reader-side barrier.

The difficulty

Dependencies must survive optimizations and function boundaries.

The outcome

Compilers used acquire.

C++26: deprecated; acquire semantics.

A useful hardware capability proved difficult to express as a practical language contract.

```cpp
struct Data { int value; int count; };

int main() {
    std::atomic<Data*> published{nullptr};
    published.store(new Data{1, 2}, std::memory_order_release);
    Data* p = published.load(std::memory_order_consume);
    std::cout << p->value << std::endl;
    delete published.load(std::memory_order_relaxed);
}
```




```asm
bl      operator new            ; x0 = new Data
ldr     d31, [x1, #3240]        ; {2 ,1} הקבוע
str     d31, [x0]               ; כתיבת value=1, count=2
add     x19, sp, #0x38          ; x19 = &published
stlr    x0, [x19]               ; published.store(p, release)    -> Store-Release
ldar    x0, [x19]               ; p = published.load(consume)    -> Load-Acquire   (1)
ldr     w1, [x0]                ; p->value: הכתובת x0 ה תוצאת-היא ldar            (2)
...
bl      ostream::operator<<(int) ; הדפסת w1
...
ldr     x0, [x19]               ; published.load(relaxed)        -> load (3)      רגיל
cbnz    x0, ...                 ; -> operator delete
```

Beyond Correctness: More Control, Same Contract

consume showed how hard it is to turn a hardware capability into a practical language contract.



Layout

How do we avoid false sharing?

C++17 · Interference sizes



Waiting

How do we wait without writing a polling loop?

C++20 · atomic::wait / notify



Access

How do we apply atomic operations to existing data?

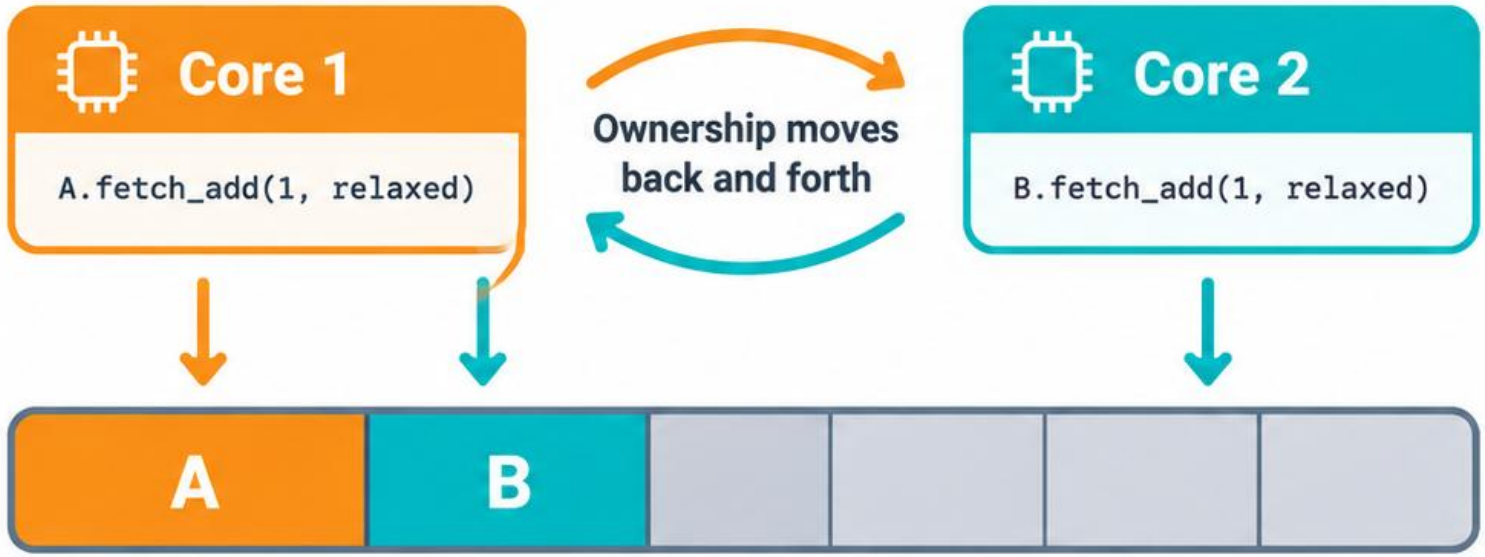
C++20 · atomic_ref

More control over layout, waiting, and access — without prescribing a CPU.

Developers express the requirements. Implementations choose the mechanisms.

False Sharing

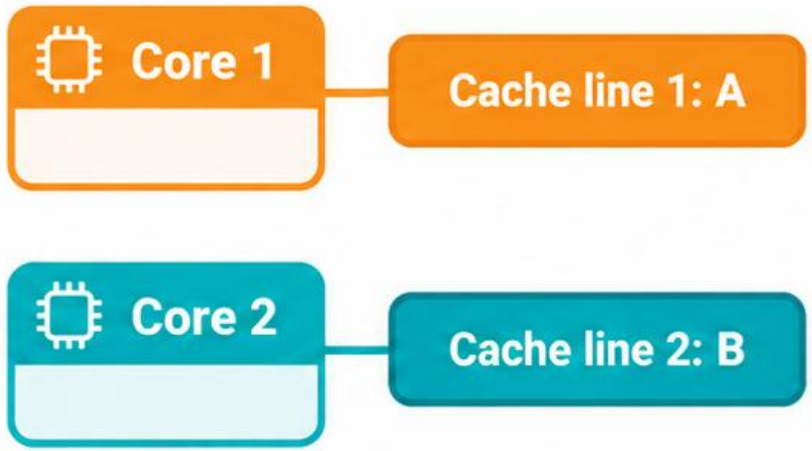
Different variables. The same cache line.



One cache line

Independent atomic counters still compete for cache-line ownership.

Separate the cache lines



No false sharing between these counters.

The code accesses different objects. Coherence operates on whole cache lines.

A performance problem — not, by itself, a data race.

```cpp
#include <atomic>
#include <cstdint>
#include <new>


struct Counters {
    alignas(std::hardware_destructive_interference_size)
    std::atomic<std::uint64_t> first{0};


    alignas(std::hardware_destructive_interference_size)
    std::atomic<std::uint64_t> second{0};
};
```

std::atomic::wait — C++20

How does a thread wait for a value to change?

Waiting thread



Thread / Core

Waits for a change

Shared atomic value



false → true

A flag or a counter

Changes the value

Updating thread



Thread / Core

One thread waits. Another thread updates.

How can we express this wait efficiently?

Waiting for a Change — Without Busy Polling

C++20: std::atomic::wait and notify

Busy polling

```
while (!ready.load(std::memory_order_acquire)) {
   // keep checking
}
```



Repeated checks consume CPU time.

Atomic waiting

```
ready.wait(false, std::memory_order_acquire);
use(data);
```



Wait while ready == false

Resume when ready changes

Express the wait; let the implementation manage it.

Writer

```
data = 42;
ready.store(true, std::memory_order_release);
ready.notify_one();
```

Notify the waiter

Store publishes. Notify wakes. Acquire provides synchronization.

Initially ready = false; one publication; no later writes to data. The waiting mechanism is implementation-dependent.

```cpp
std::mutex m;
std::condition_variable cv;
bool ready = false;

//------------ Consumer
{
    std::unique_lock lk(m);
    cv.wait(lk, [] { return ready; });
}

//------------ Producer
{
    std::lock_guard lk(m);
    ready = true;
}
cv.notify_one();
```

Before C++20

```cpp
// Writer
data = 42;
ready.store(true, std::memory_order_release);
ready.notify_one();


// Reader
ready.wait(false, std::memory_order_acquire);
use(data);
```

Since C++20

std::atomic_ref — Atomic Access to Existing Storage

C++20: Keep the object. Change how you access it.

```
alignas(std::atomic_ref<int>::required_alignment)
int value = 0;
```

The same underlying object in every phase

1. Exclusive phase

One thread

```
value = 10;
```

Ordinary access

Synchronize phase transition

2. Concurrent phase

Two threads

```
std::atomic_ref<int> ref{value};
ref.fetch_add(1,
      std::memory_order_relaxed);
```

Access only through atomic_ref

Finish workers; destroy all refs

3. Exclusive phase

One thread

```
int result = value;
```

Ordinary access resumes

⚠ Scope alone does not synchronize threads.

Respect alignment and object lifetime.
While atomic_refs exist, access the object only through atomic_refs.

Schematic phases: transitions require synchronization; no overlapping ordinary and atomic access.

```cpp
struct Edge { int from, to; };

std::vector<int> build_offsets(const std::vector<Edge>& edges, int num_vertices){
    static_assert(std::atomic_ref<int>::required_alignment == alignof(int));

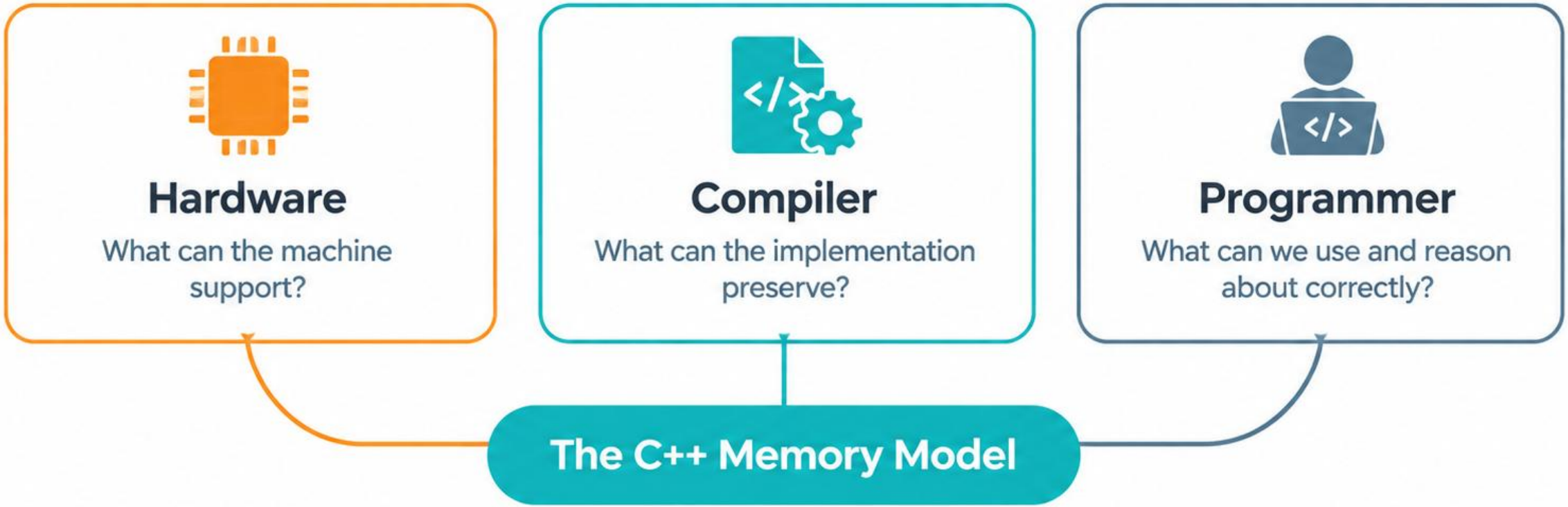
    // Phase 1: plain ints - copyable, resizable, contiguous
    std::vector<int> degree(num_vertices, 0);

    // Phase 2: parallel - the only place that needs atomicity
    std::for_each(std::execution::par, edges.begin(), edges.end(),
        [&degree](const Edge& e) {
            std::atomic_ref<int>(degree[e.to])
                .fetch_add(1, std::memory_order_relaxed);
        });

    // Phase 3: plain ints again - vectorizable, zero atomic overhead
    std::vector<int> offsets(num_vertices + 1, 0);
    std::inclusive_scan(degree.begin(), degree.end(), offsets.begin() + 1);
    return offsets;
}
```

One Contract. Three Requirements.

The lesson from memory_order_consume



A useful hardware capability is not enough: the contract must also be implementable and usable.

This tension continues to shape the memory model.

C++26: Simplifying the Memory Model

Before	consume promised a cheaper acquire, but needed a whole dependency model.
The problem	Compilers never implemented it. They all treated consume as acquire.
C++26	**consume now means acquire. Your code does not change.**

Source: P3475R2 | C++ draft, Annex D

C++26: From Memory Ordering to Asynchronous Composition

std::execution — a framework for describing and composing asynchronous work

So far: memory accesses

Write → Ordering & synchronization → Read

Which relationships make the program correct?

Next: asynchronous work

1. Where does the work run?
2. How does one stage pass its result to the next?
3. How do we compose operations?

Produce → Result → Transform → Result → Consume

Execution resources

- Thread pool
- I/O context
- Accelerator*

Separate what runs from where it runs

A new execution framework — not a replacement memory model.

*Accelerator support depends on the scheduler and implementation.

std::execution: Three Core Abstractions

Describe work. Receive completion. Choose an execution resource.

Sender

Describes an asynchronous operation and its completion possibilities.

Receiver

Receives completion:

- Value
- Error
- Stopped

Scheduler

Schedules work on an execution resource.

Composable work

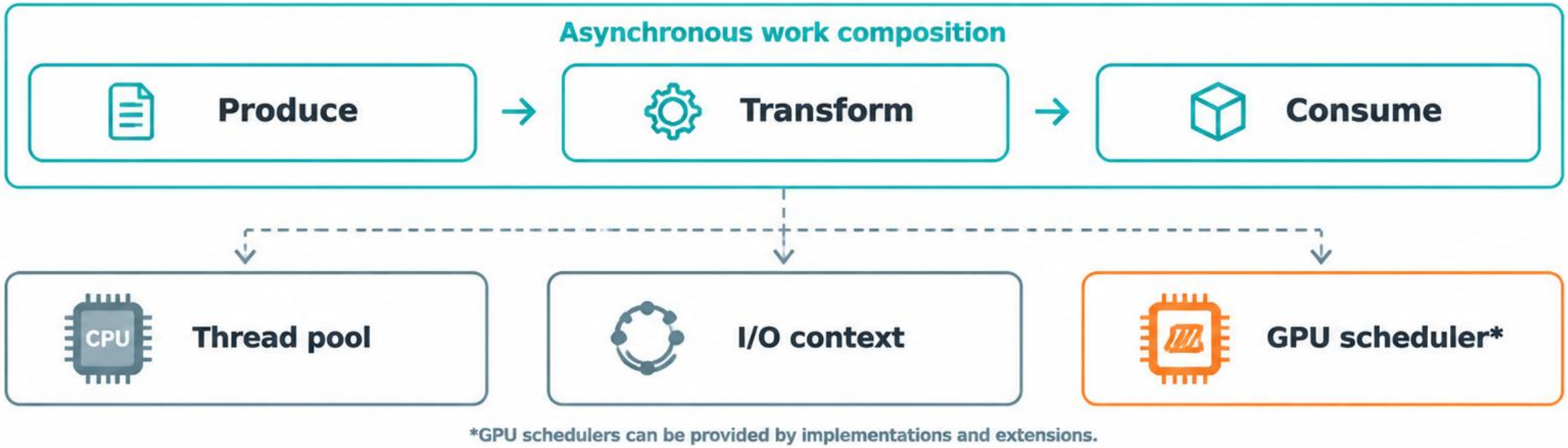


Build with reusable abstractions instead of wiring each step by hand.

Compose asynchronous work into pipelines and graphs.

Composable Work. Target-Specific Responsibilitioes.

std::execution separates work composition from execution resources



*GPU schedulers can be provided by implementations and extensions.

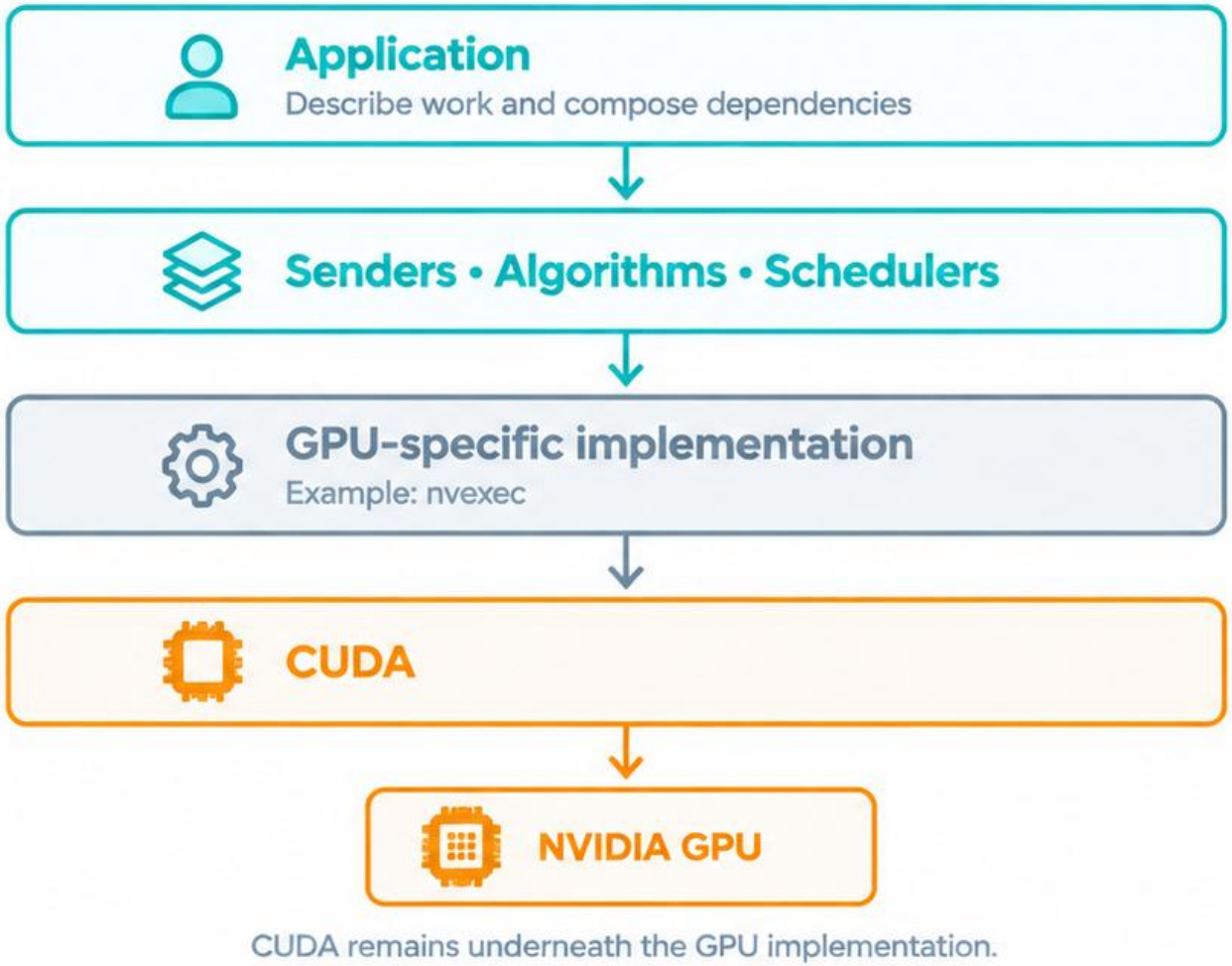
The target still matters

- Memory access
- Data transfer
- Correctness

Composition is portable. Target requirements do not disappear.

std::execution: An Abstraction Above CUDA

Describe and compose asynchronous work — let the implementation manage execution.



CUDA remains underneath the GPU implementation.

What the abstraction provides

A common language for asynchronous composition

Resource selection through schedulers

What still matters

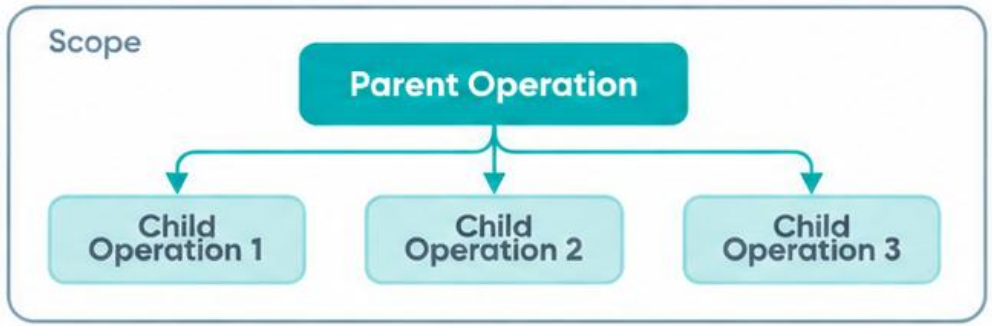
- GPU-compatible code
- Memory access and data transfer
- Object lifetimes and correctness

An abstraction over execution mechanisms — not a replacement for CUDA.

Why This Is an Architectural Upgrade

Structured composition + low-overhead abstractions

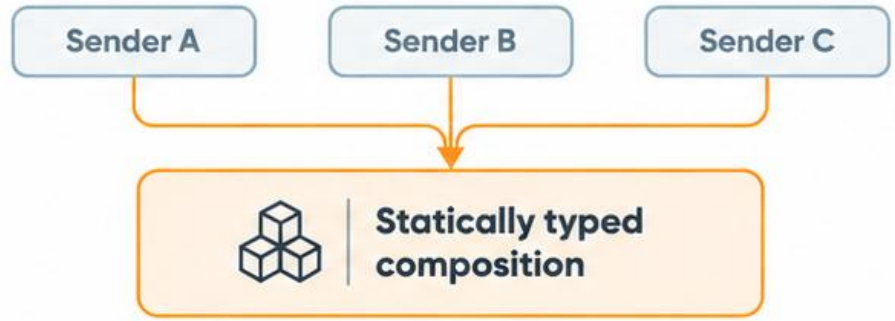
Structured Concurrency



- Compose child operations with when_all
- Wait for every child before completing
- Propagate cooperative stop requests

A stop request is not forced termination.

Low-Overhead Composition



- Work graphs represented in types
- Operation states can embed child states
- Enables inlining and fewer allocations

No universal guarantee of zero allocations or full inlining.

From manual coordination to explicit composition

Make dependencies, completion and cancellation part of the design.

More explicit lifetimes. Less coordination boilerplate. Optimizable composition.

```cpp
namespace ex = stdexec;

const std::thread::id main_id = std::this_thread::get_id();

int main() {
    // Obtain a scheduler for a parallel execution resource.
    ex::scheduler auto sch = ex::get_parallel_scheduler();
    // Describe the work — nothing starts yet.
    ex::sender auto work =
        ex::schedule(sch)
      | ex::then([] { return 21; })
      | ex::then([](int x) { return x * 2; });
    // Start the work and wait for completion.
    auto [result] = ex::sync_wait(std::move(work)).value();
    std::println("result = {}", result);  // 42
}
```

```cpp
namespace ex = stdexec;

exec::static_thread_pool cpu_context{4};
auto cpu = cpu_context.get_scheduler();

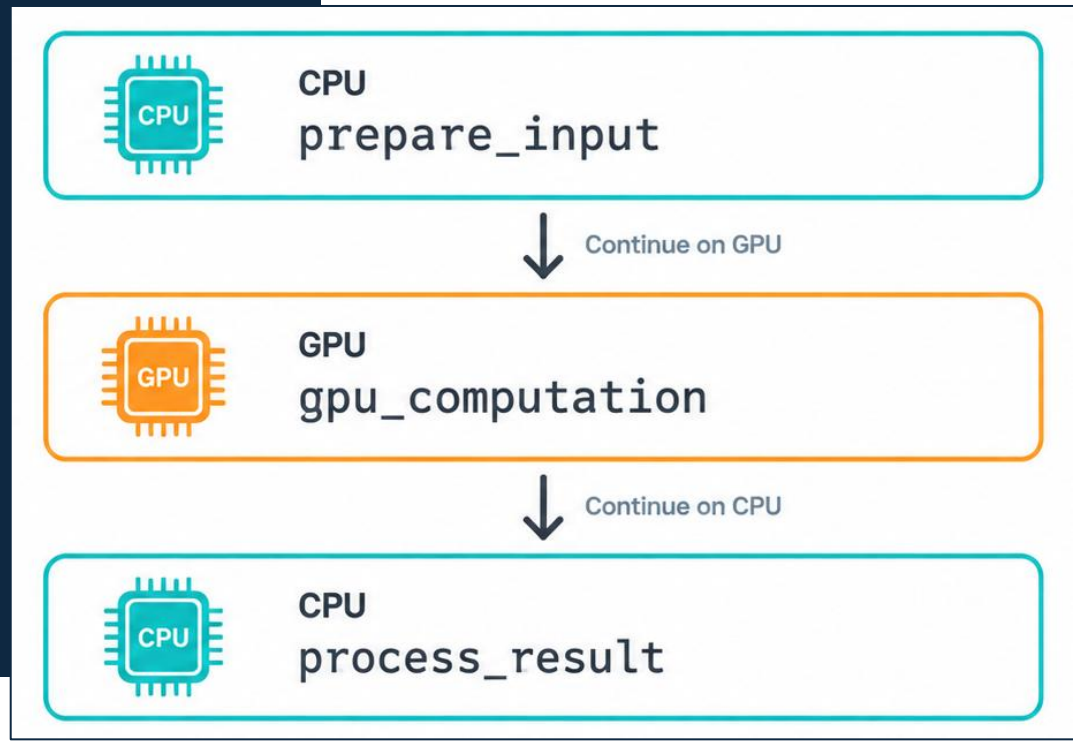
nvexec::stream_context gpu_context;
auto gpu = gpu_context.get_scheduler();

// Conceptual sketch — not a complete runnable GPU example.
auto work =
    ex::schedule(cpu)
    | ex::then(prepare_input)

    | ex::continues_on(gpu)
    | ex::then(gpu_computation)

    | ex::continues_on(cpu)
    | ex::then(process_result);

ex::sync_wait(std::move(work));
```



Phase 1: With std::async and std::future

```
std::future<double> compute_total(int id) {
    return std::async(std::launch::async, [id] {
        User u = load_user(id);

        auto orders_f   = std::async(std::launch::async, load_orders,   u);
        auto discount_f = std::async(std::launch::async, load_discount, u);

        auto   orders   = orders_f.get();
        double discount = discount_f.get();

        return sum(orders) * (1.0 - discount);
    });
}
```



```cpp
using ErrorCb = std::function<void(std::exception_ptr)>;

template <class F, class OnOk>
void run_async(F work, OnOk ok, ErrorCb err) {
    std::thread([=] {
        try { ok(work()); }
        catch (...) { err(std::current_exception()); }
    }).detach();
}

void compute_total_async(int id, std::function<void(double)> ok, ErrorCb err) {
    run_async([id] { return load_user(id); },
        [ok, err](User u) {   // <- nesting level 1
            struct State {
                std::mutex m;
                std::optional<std::vector<Order>> orders;
                std::optional<double> discount;
                std::exception_ptr error;
                bool done = false;
            };
            auto st = std::make_shared<State>();
            auto finish = [st, ok, err](auto update) {
                std::unique_lock lock(st->m);
                if (st->done) return;  // already finished / failed
                update(*st);
                if (st->error) {
                    st->done = true;
                    auto e = st->error;
                    lock.unlock();
                    err(e);
                } else if (st->orders && st->discount) {
                    st->done = true;
                    double total = sum(*st->orders) * (1.0 - *st->discount);
                    lock.unlock();
                    ok(total);
                }
            };
            run_async([u] { return load_orders(u); },  // <- nesting level 2
                [finish](std::vector<Order> o) { finish([&](State& s) {
                                                 s.orders = std::move(o); }); },
                [finish](std::exception_ptr e) { finish([&](State& s) {
                                                 s.error = e; }); });

            run_async([u] { return load_discount(u); }, // <- nesting level 2
                [finish](double d)             { finish([&](State& s) {
                                                 s.discount = d; }); },
                [finish](std::exception_ptr e) { finish([&](State& s) {
                                                 s.error = e; }); });
        },
        err);
}

void run(int id) {
    std::println("\n=== compute_total_async({}) ===", id);
    std::promise<void> done;  // only so main doesn't exit early
    compute_total_async(id,
        [&](double total) { log("total = {}", total); done.set_value(); },
        [&](std::exception_ptr e) {
            try { std::rethrow_exception(e); }
            catch (const std::exception& ex) { log("error: {}", ex.what()); }
            done.set_value();
        });
    log("main thread is free to do other work...");
    done.get_future().wait();
}

int main() {
    std::println("Stage 2: callbacks");
    run(7);
    run(-1);  // error path
}
```



Phase 3: With std::execution

```
#include <execution>
namespace ex = std::execution;

ex::sender auto compute_total(ex::scheduler auto sch, int id) {
    return ex::starts_on(sch, ex::just(id))
         | ex::then(load_user)
         | ex::let_value([sch](User& u) {
             return ex::when_all(
                 ex::starts_on(sch, ex::just(u) | ex::then(load_orders)),
                 ex::starts_on(sch, ex::just(u) | ex::then(load_discount)));
           })
         | ex::then([](std::vector<Order> orders, double discount) {
             return sum(orders) * (1.0 - discount);
           });
}
```

```cpp
int main() {
    auto sch = ex::get_parallel_scheduler();

    try {
        auto [total] = std::this_thread::sync_wait(compute_total(sch, 7)).value();
        std::println("total = {}", total);
    } catch (const std::exception& e) {
        std::println("error: {}", e.what());
    }
}
```



Bonus: With coroutines

```
ex::task<double> compute_total_co(ex::parallel_scheduler sch, int id) {
    User u = co_await ex::starts_on(sch, ex::just(id) | ex::then(load_user));

    auto [orders, discount] = co_await ex::when_all(
        ex::starts_on(sch, ex::just(u) | ex::then(load_orders)),
        ex::starts_on(sch, ex::just(u) | ex::then(load_discount)));

    co_return sum(orders) * (1.0 - discount);
}
// main:
// auto [total] = std::this_thread::sync_wait(compute_total_co(sch, 7)).value();
```



Different Layers. Different Responsibilities.

Complementary abstractions — not replacements for one another



Memory Model

Ordering and validity of memory accesses

Which accesses are correctly ordered?



Execution Model

Composition and execution-resource selection

How is work connected, and where does it run?



Coroutines

Suspendable control flow

How do we express suspension and continuation?

Three perspectives on the same computation

Better abstractions do not remove lower-level contracts.

What Should We Take Away?

Three principles for correct, composable, portable concurrency

01

Reason about guarantees — not assumed instruction order.

Prove correctness from the contract, not from hardware intuition.



02

Compose dependencies — avoid hand-written coordination where possible.

Express relationships directly instead of wiring every continuation by hand.



03

Separate computation from execution — without ignoring target constraints.

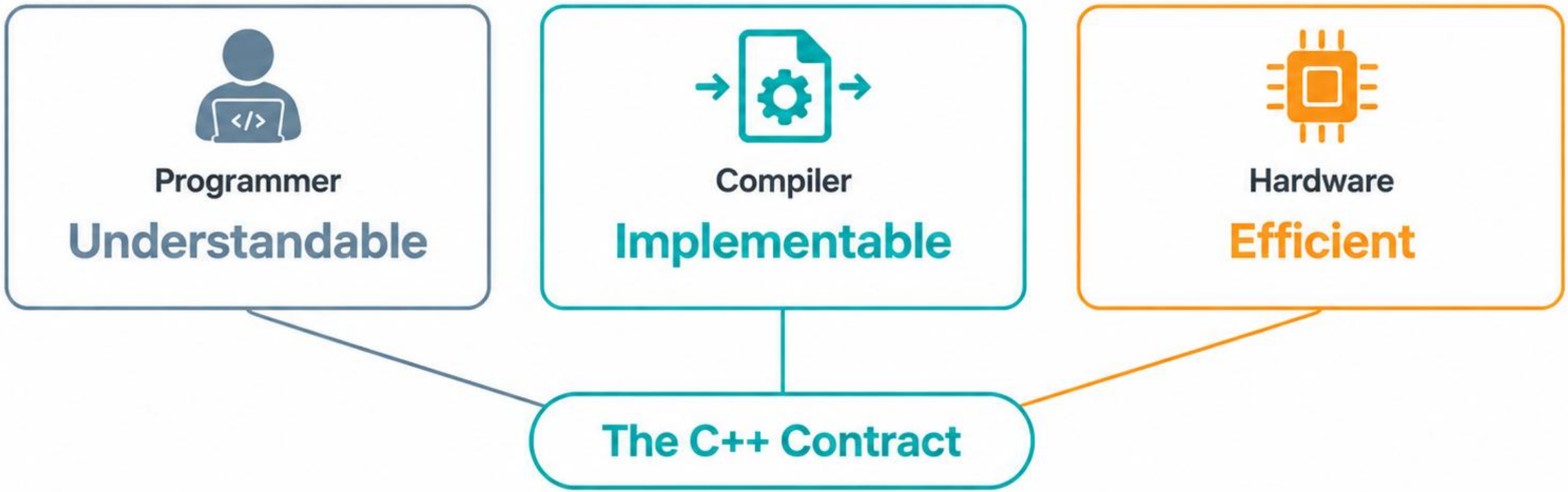
Choose execution resources explicitly; respect code and memory requirements.



Express the required relationships. Preserve the underlying contracts.

The Contract Still Has *Three Parties.*

Correctness, implementability, and efficiency must meet.



Strong enough for correctness. Flexible enough for efficient execution.

Express the required relationships. Let the implementation realize them.

Thank You!

Questions & Discussion

