

The C++ Execution Model

Bryce Adelstein Lelbach

 @blelbach

Core C++ 2019



The Abstract Machine



The C++ abstract machine is a portable abstraction of your operating system, kernel and hardware.

The Abstract Machine



The C++ abstract machine is a portable abstraction of your operating system, kernel and hardware.

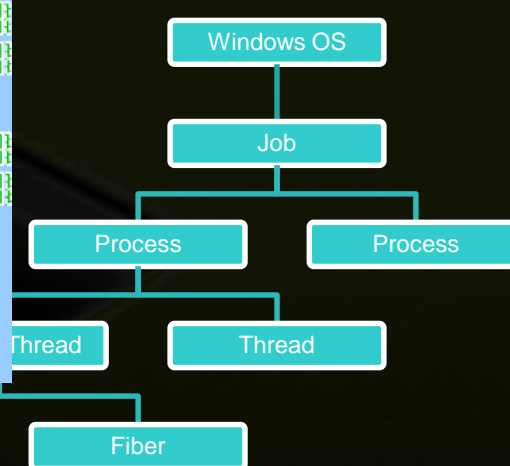
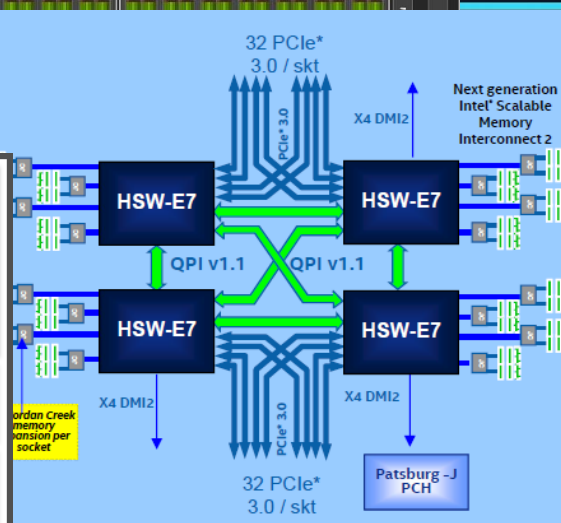
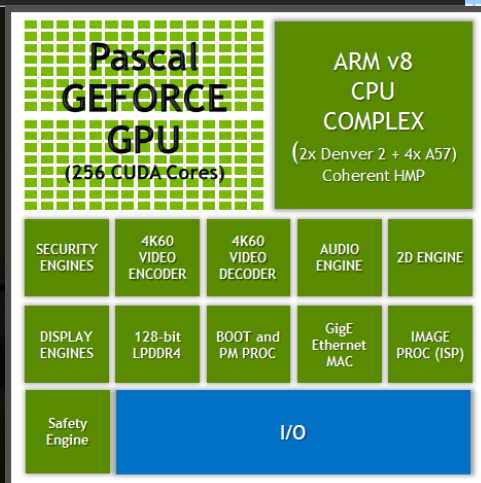
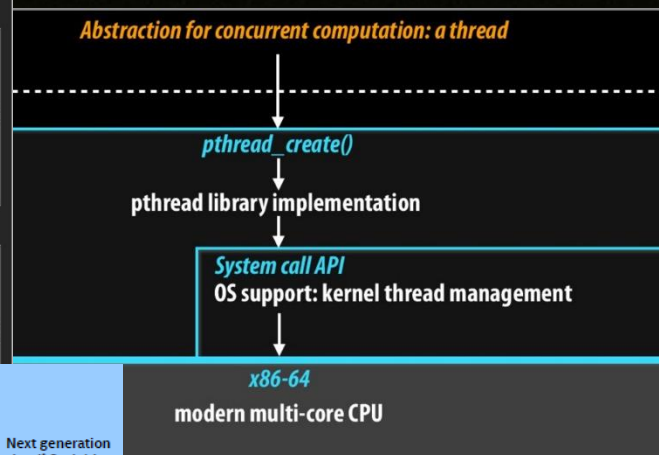
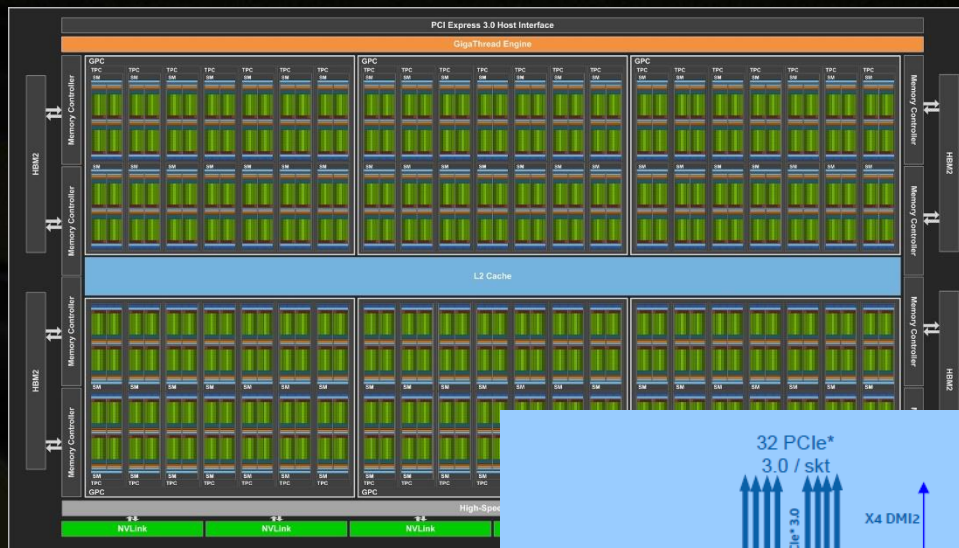
The abstract machine is the intermediary between your C++ program and the system that it is run on.

The Abstract Machine



```
int main()  
{  
    std::thread t(f);  
  
    // ...  
}
```


The Abstract Machine



The Abstract Machine



Portable Code

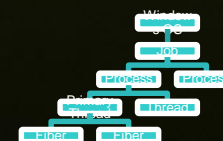
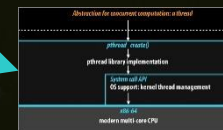
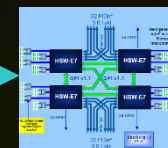
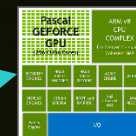
Abstract Machine

Concrete Systems

```
int main()
{
    std::thread t(f);
    // ...
}
```

Threads

Storage



The Abstract Machine



C++ programs describe operations that are performed on the abstract machine.

C++ implementations define the characteristics of the abstract machine and translate operations on the abstract machine into operations on the system.

The Abstract Machine



Threads



Storage

Storage Model



Storage is flat; no notion hierarchy (caches, etc).

Objects reside in storage at a single memory location.

[\[intro.object\] p9](#)

Storage Model



Some objects may not have a unique memory location.

- **Eligible empty base classes.**
- **Objects marked `[[no_unique_address]]`.**

[\[intro.object\] p7](#)

Storage Model



An implementation is allowed to store two objects at the same machine address or not store an object at all.

[\[basic.memobj\] footnote 32](#)

Storage Model



An object cannot have more than one memory location.

Storage Model



```
struct A {  
    X x;  
  
    X& X::operator=(X const& rhs)  
    {  
        if (&rhs == this) return *this;  
        auto newx = new X(*rhs.x);  
        delete x;  
        x = newx;  
        return *this;  
    }  
};
```

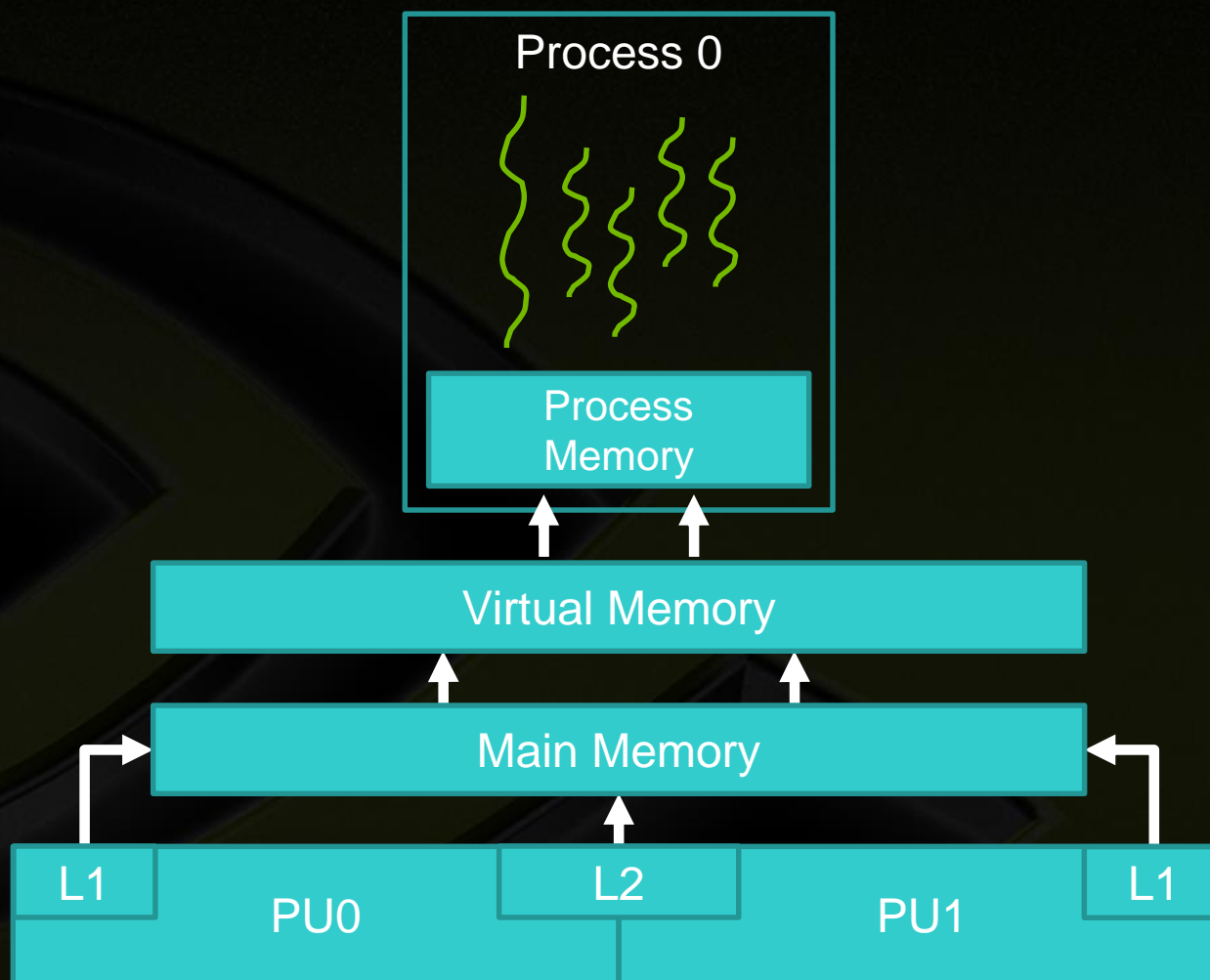
Storage Model



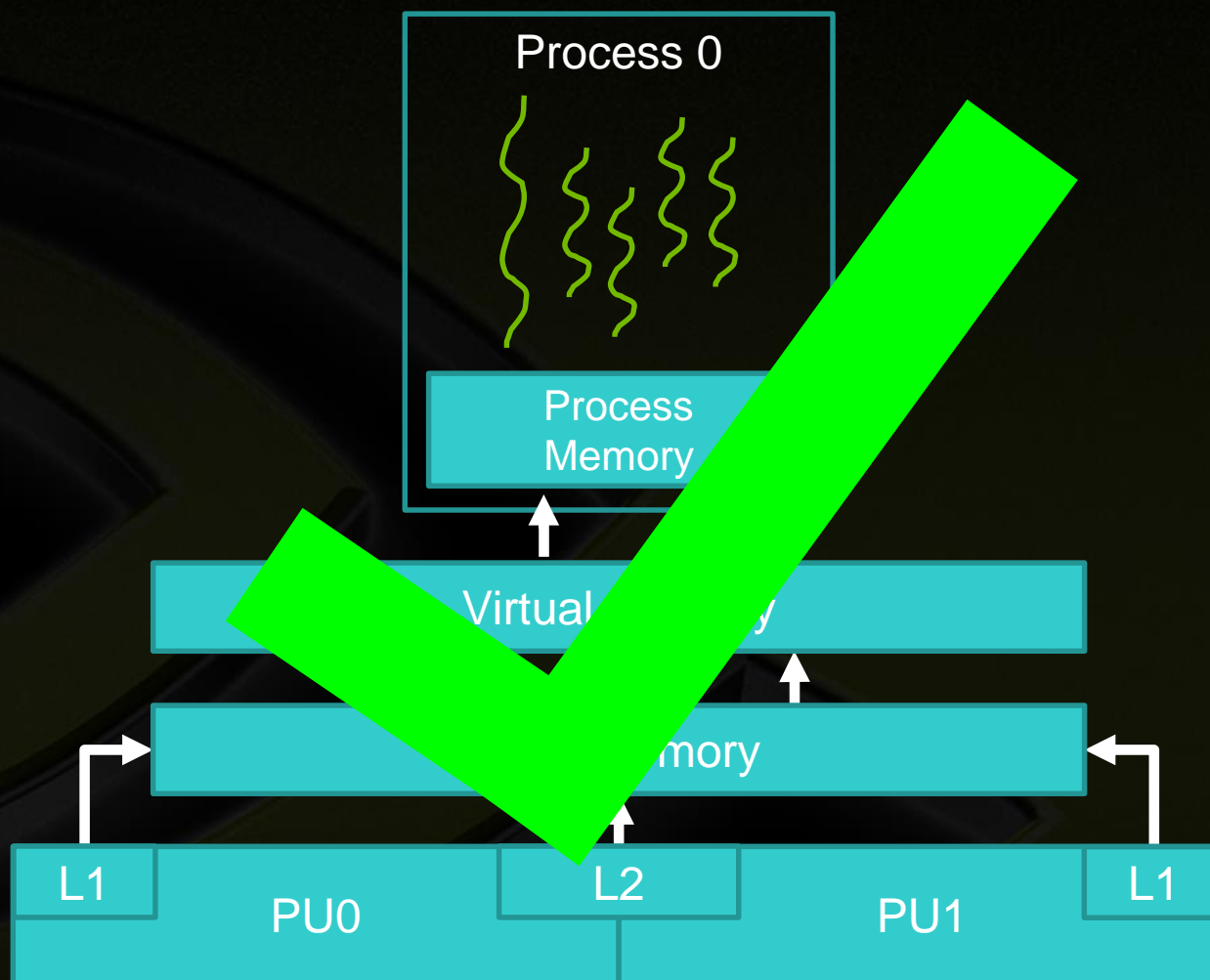
Every thread in a program can potentially access every object and function in a program.

[\[intro.multithread\] p1 s2](#)

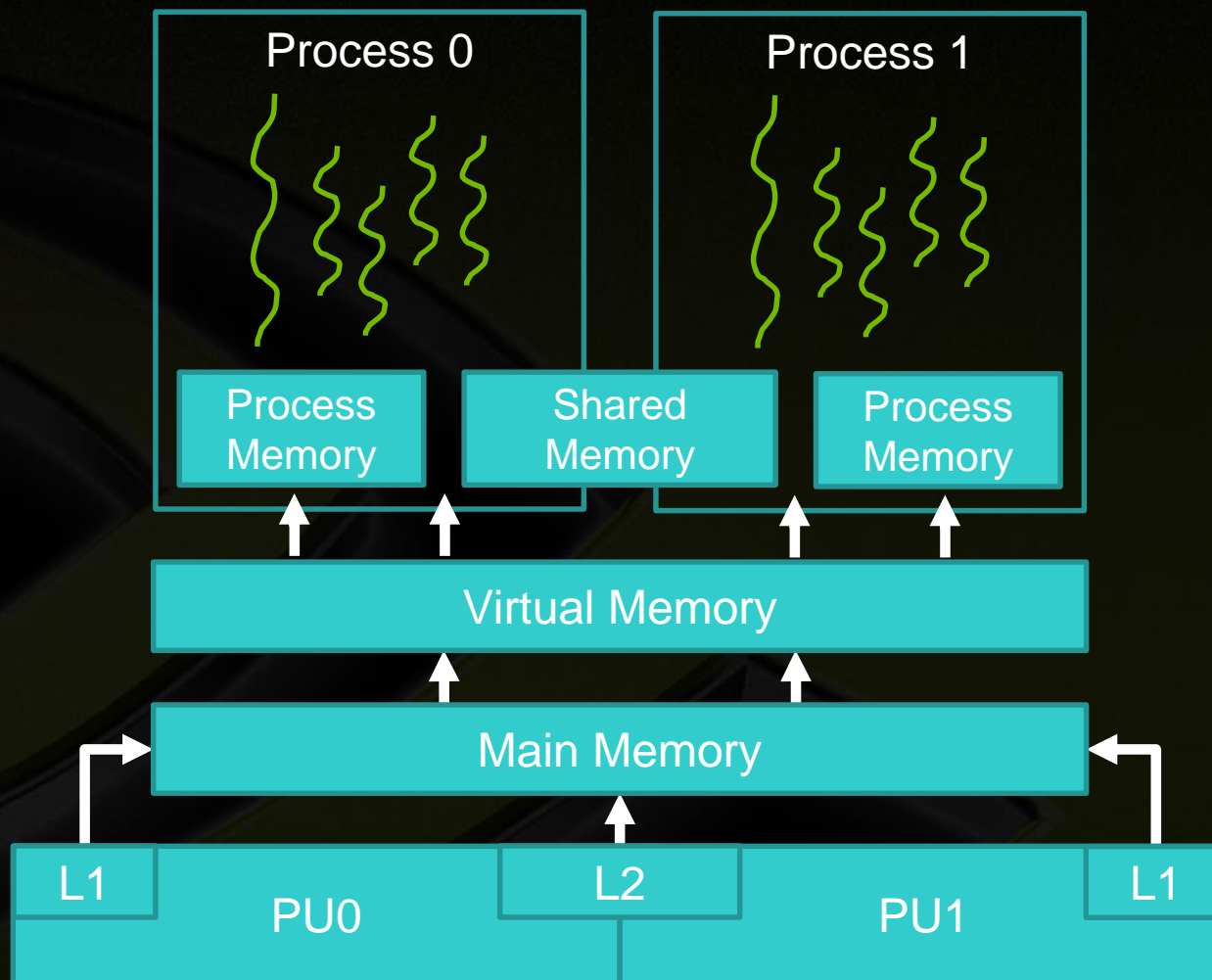
Storage Model



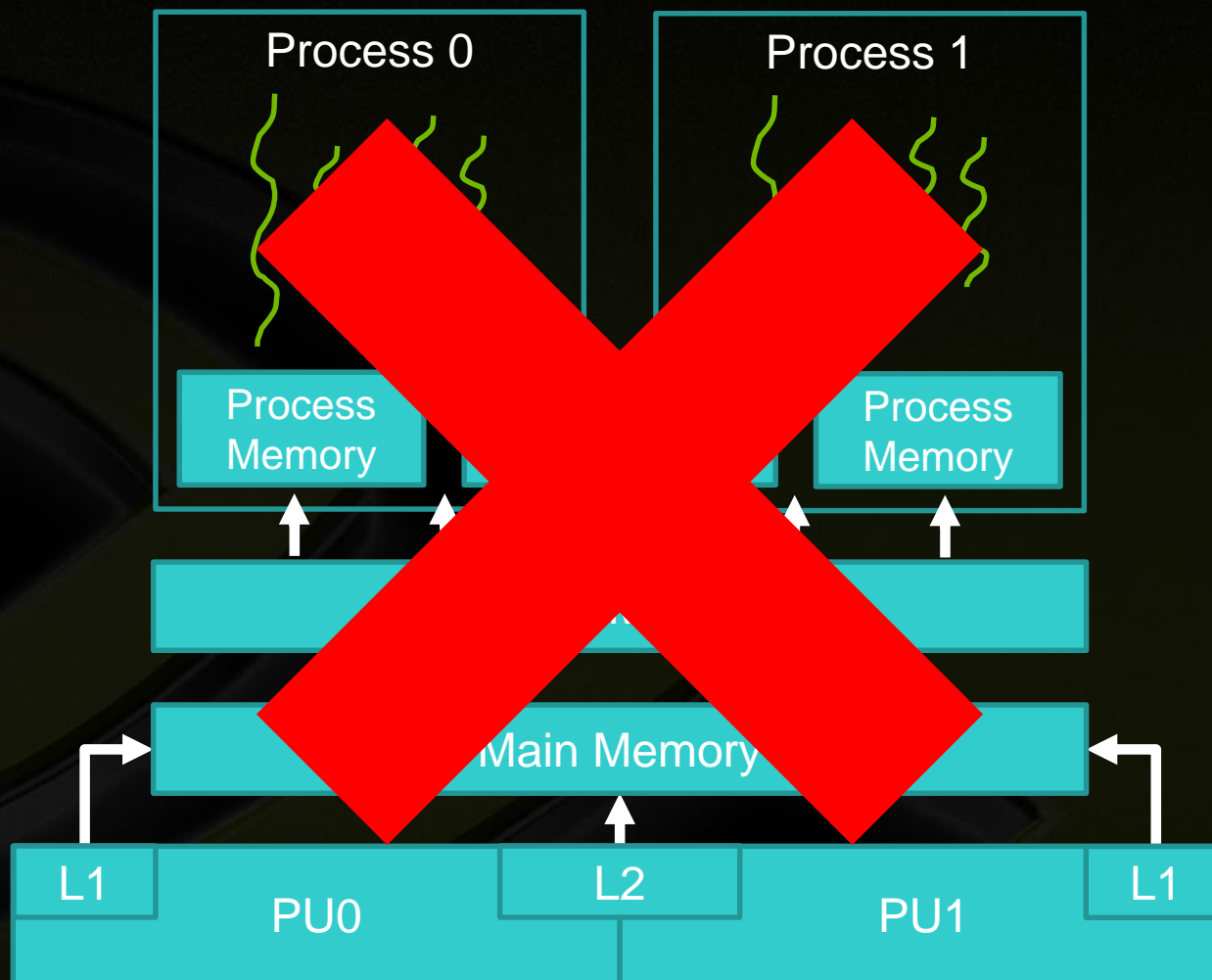
Storage Model



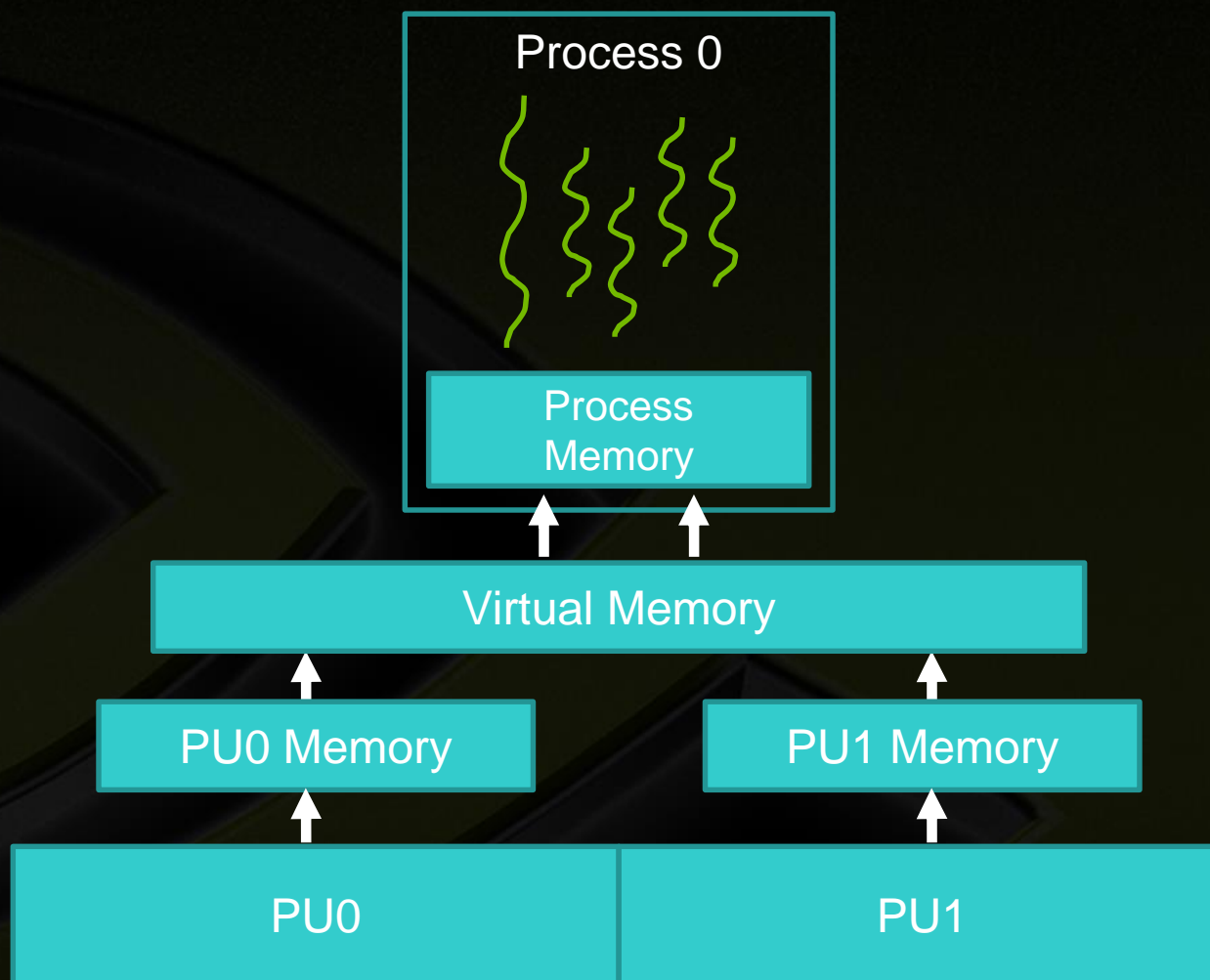
Storage Model



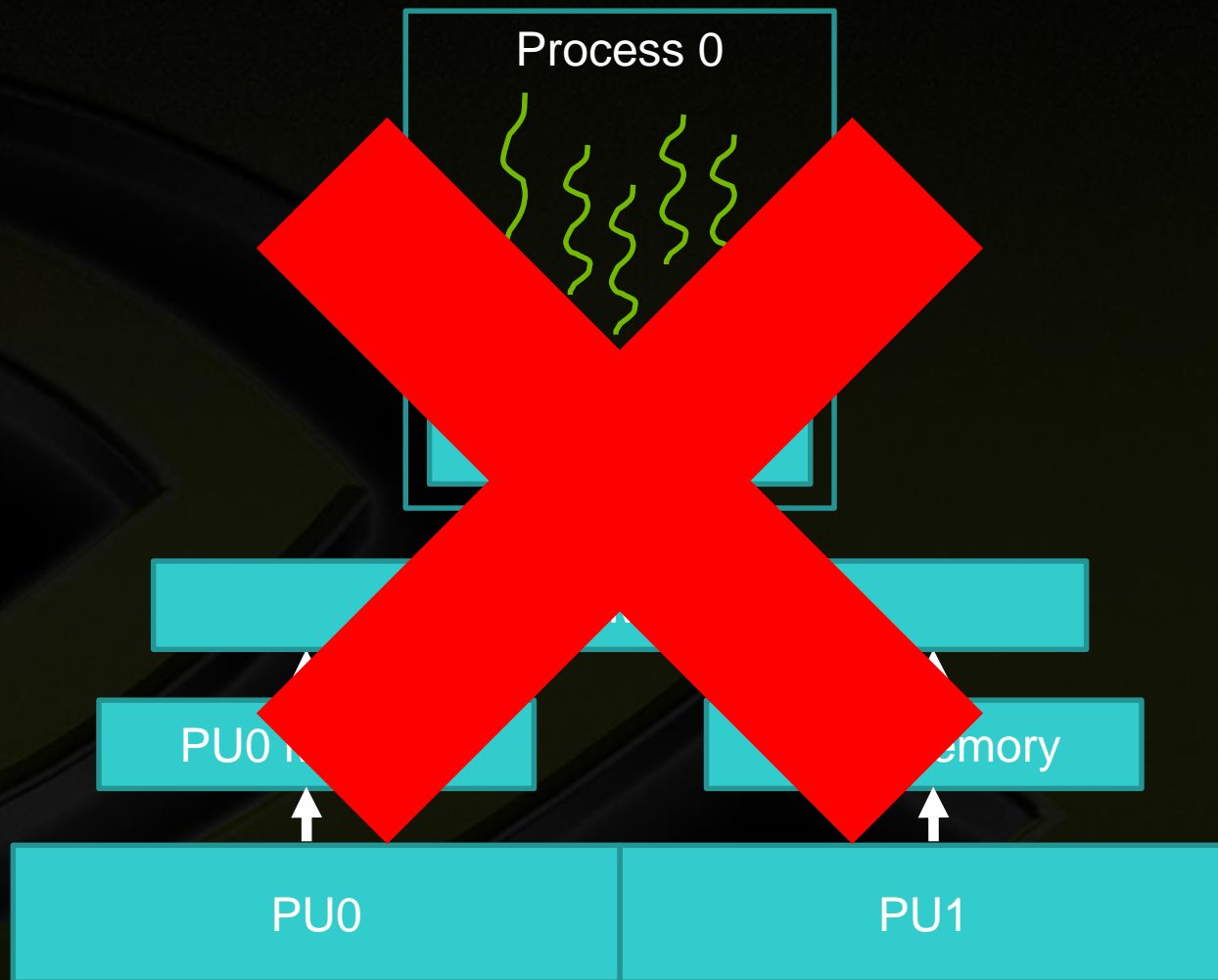
Storage Model



Storage Model



Storage Model



Threads of Execution



A *thread of execution* is a single flow of control in a program which evaluates a function call; threads may run concurrently.

[\[intro.multithread\] p1 s1](#)

Threads of Execution



A **thread of execution** is a single flow of control in a program which evaluates a function call; threads may run concurrently.

[\[intro.multithread\] p1 s1](#)

Main Thread of Execution

Evaluate `main()`

Threads of Execution



A **thread of execution** is a single flow of control in a program which evaluates a function call; threads may run concurrently.

[\[intro.multithread\] p1 s1](#)

Main Thread of Execution

```
std::thread t(f);
```

Evaluate `main()`

Evaluate `f()`

Threads of Execution



Variables with static storage duration are initialized as a consequence of program initiation. Variables with thread storage duration are initialized as a consequence of thread execution.

[\[basic.start.static\] p1](#)

Main Thread of Execution

`std::thread t(f);`

Static storage initialization

Evaluate `main()`

Evaluate `f()`

Static storage destruction

Threads of Execution



Variables with static storage duration are initialized as a consequence of program initiation. Variables with thread storage duration are initialized as a consequence of thread execution.

[\[basic.start.static\] p1](#)

Main Thread of Execution

Static storage initialization

Thread storage initialization

Evaluate `main()`

Thread storage destruction

Static storage destruction

```
std::thread t(f);
```

Thread storage initialization

Evaluate `f()`

Thread storage destruction

Threads of Execution



Okay, so threads **evaluate** a function call.

What does it mean to **evaluate** a function call?

Expressions



An expression is a sequence of operators and operands that specifies a computation.

[\[expr.pre\] p1 s2](#)

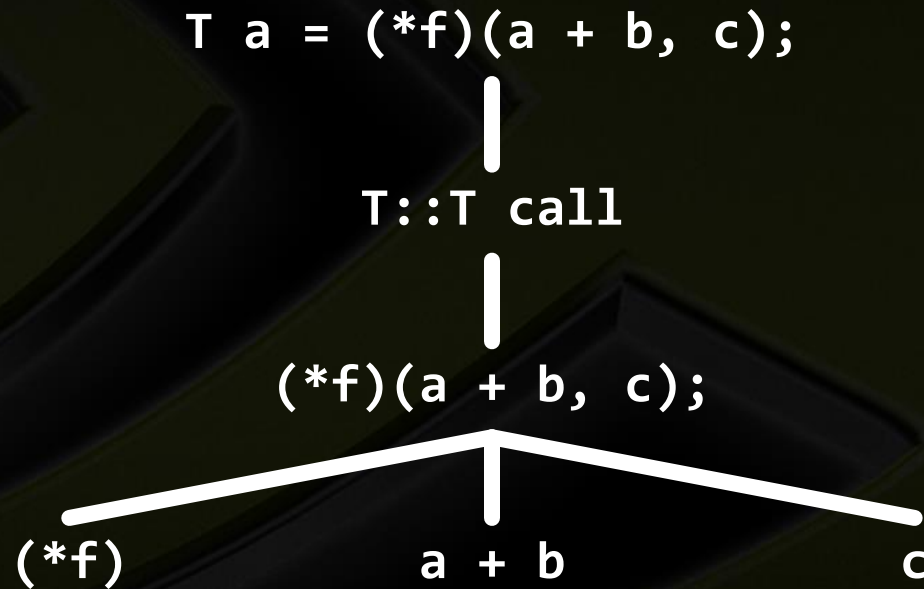
```
f();  
f(a, b);  
  
a + b;  
// `operator+(a, b)` call.  
  
T a = 2;  
T a(2);  
// `T::T(int)` call.
```

Expressions



Subexpressions are a part of a larger **expression**.

[\[intro.execution\] p3, p4](#)



Expressions



Full expressions are not **subexpressions**.

[\[intro.execution\] p5](#)

```
if (a == T()) { ... }  
// Full expression includes:  
// lvalue-to-rvalue conversion  
// T-to-bool conversion  
// operator==(T, T) call
```

Expressions



Full expressions are not subexpressions.

[\[intro.execution\] p5](#)

```
if (a == T()) { ... }  
// Full expression includes:  
// lvalue-to-rvalue conversion  
// T-to-bool conversion  
// operator==(T, T) call  
  
{  
    T b;  
} // Full expression: T::~~T
```


Expressions



Full expressions may include subexpressions that are not lexically part of the expression.

[\[intro.execution\] p6](#)

```
void f(T a = g());  
  
f();  
// Full expression includes:  
// g call  
// T::T call  
// f call
```

Evaluations



Evaluation of an **expression** includes **value computations** and the initiation of **side effects**.

[intro.execution] p7 s2

Evaluations



Side effects change the environment:

Evaluations



Side effects change the environment:

- Reading a volatile object or modifying any object.

Evaluations



Side effects change the environment:

- Reading a volatile object or modifying any object.
- Calling a library I/O function.

Side effects change the environment:

- Reading a volatile object or modifying any object.
- Calling a library I/O function.
- Calling a function that does any of the above.

[\[intro.execution\] p7 s1](#)

Evaluations



```
int a;  
int b;  
a = a + b;  
  
cout << a * a;  
  
foo(a + b);
```

Evaluations



Value computations are pure and have no observable effect.

[intro.execution] p7 s2

Evaluations



Completion of the **execution** of an **evaluation** does not imply completion of its **side effects**.

[\[intro.execution\] p7 s3](#)

Evaluations



```
cout << a * a;
```

Sequenced Before



Given any two **evaluations** A and B within the same thread of execution, if A is **sequenced before** B, then the **execution** of A shall precede the **execution** of B.

[\[intro.execution\] p8 s2](#)

Sequenced Before



The **sequenced before** relationship is...

- **Asymmetric:** A is **sequenced before** B does not imply that B is **sequenced before** A.

[\[intro.execution\] p8 s1](#)

- **Transitive:** If A is **sequenced before** B and B is **sequenced before** C, then A is **sequenced before** C.

[\[intro.execution\] p8 s1](#)

Sequenced Before



Each **full expression** is **sequenced before** the next **full expression** in program order.

[\[intro.execution\] p9](#)

```
a;  
b;  
// a sequenced before b
```

Sequenced Before



If A and B are indeterminately sequenced, then either A is **sequenced before** B or B is **sequenced before** A, but it is unspecified which. E.g. A and B are not **interleaved**.

[\[intro.execution\] p8 s4](#)

Sequenced Before



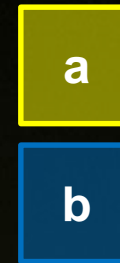
If A and B are unsequenced, then A is not sequenced before B and B is not sequenced before A. E.g. A and B may be interleaved.

[\[intro.execution\] p8 s3](#)

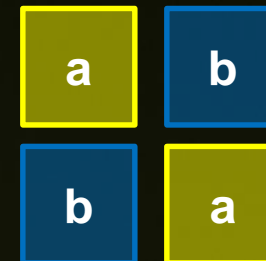
Sequenced Before



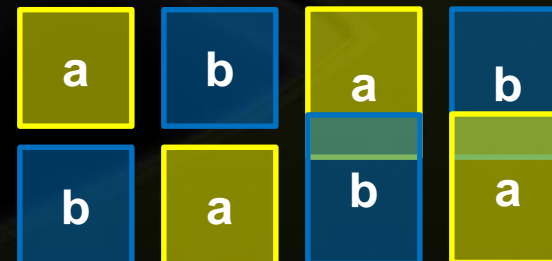
A is sequenced before B



A and B are
indeterminately sequenced



A and B are
unsequenced



Sequenced Before



```
constexpr float a = // ...  
std::vector<float> x = // ...  
std::vector<float> y = // ...
```

```
std::for_loop(  
    std::execution::seq,  
    0, x.size(),  
    [&] (int i) {  
        y[i] += a * x[i];  
    }  
);
```

```
load y[i]  
load x[i]  
fma a * x[i] + y[i]  
store y[i]  
load y[i+1]  
load x[i+1]  
fma a * x[i+1] + y[i+1]  
store y[i+1]  
load y[i+2]  
load x[i+2]  
fma a * x[i+2] + y[i+2]  
store y[i+2]  
load y[i+3]  
load x[i+3]  
fma a * x[i+3] + y[i+2]  
store y[i+3]
```

Sequenced Before



```
constexpr float a = // ...  
std::vector<float> x = // ...  
std::vector<float> y = // ...
```

```
std::for_loop(  
    std::execution::unseq,  
    0, x.size(),  
    [&] (int i) {  
        y[i] += a * x[i];  
    }  
);
```

```
load y[i]  
load y[i+1]  
load y[i+2]  
load y[i+3]  
load x[i]  
load x[i+1]  
load x[i+2]  
load x[i+3]  
fma a * x[i] + y[i]  
fma a * x[i+1] + y[i+1]  
fma a * x[i+2] + y[i+2]  
fma a * x[i+3] + y[i+2]  
store y[i]  
store y[i+1]  
store y[i+2]  
store y[i+3]
```


Sequenced Before



```
constexpr float a = // ...  
std::vector<float> x = // ...  
std::vector<float> y = // ...
```

```
std::for_loop(  
    std::execution::unseq,  
    0, x.size(),  
    [&] (int i) {  
        y[i] += a * x[i];  
    }  
);
```

```
vload y[i:i+3]  
vload x[i:i+3]  
vfma a * x[i:i+3] + y[i:i+3]  
vstore y[i:i+3]
```

Statements



Statements are compositions of full expressions.

Statements



Statements are compositions of full expressions.

```
{  
    statement0;  
    statement1;  
    // `statement0` is  
    // sequenced before  
    // `statement1`.  
}
```

Statements



Statements are compositions of full expressions.

```
{  
    statement0;  
    statement1;  
    // `statement0` is  
    // sequenced before  
    // `statement1`.  
}
```

```
if (condition)  
    body;  
// `condition` is  
// sequenced before  
// `body`.
```

Statements



Statements are compositions of full expressions.

```
{  
    statement0;  
    statement1;  
    // `statement0` is  
    // sequenced before  
    // `statement1`.  
}
```

```
if (condition)  
    body;  
// `condition` is  
// sequenced before  
// `body`.
```

```
while (condition)  
    body;  
// Each evaluation of `condition`  
// is sequenced before each  
// evaluation of `body`.
```

Function Evaluation



When calling a **function**...

1. Every **evaluation** within the function and every **evaluation** not within the function are **indeterminately sequenced**.

[\[intro.execution\] p11 s2](#)

Function Evaluation



When calling a **function**...

1. Every **evaluation** within the function and every **evaluation** not within the function are **indeterminately sequenced**.

[\[intro.execution\] p11 s2](#)

2. The **expression** designating the function is **sequenced before** the argument **expressions**.

[\[expr.call\] p8 s1](#)

Function Evaluation



When calling a **function**...

1. Every **evaluation** within the function and every **evaluation** not within the function are **indeterminately sequenced**.

[\[intro.execution\] p11 s2](#)

2. The **expression** designating the function is **sequenced before** the argument **expressions**.

[\[expr.call\] p8 s1](#)

3. Each argument **expression** is **indeterminately sequenced** with all other argument **expressions**.

[\[expr.call\] p8 s2](#)

Function Evaluation



When calling a **function**...

1. Every **evaluation** within the function and every **evaluation** not within the function are **indeterminately sequenced**.

[\[intro.execution\] p11 s2](#)

2. The **expression** designating the function is **sequenced before** the argument **expressions**.

[\[expr.call\] p8 s1](#)

3. Each argument **expression** is **indeterminately sequenced** with all other argument **expressions**.

[\[expr.call\] p8 s2](#)

4. Every **expression** in the body of the function is **sequenced after** the **expression** designating the **function** and every argument **expression** of the **function** .

[\[intro.execution\] p11 s1](#)

Function Evaluation



```
void b(...) { e; }
```

```
g(a, (b)(c, d), f);
```

Function Evaluation



```
void b(...) { e; }
```

```
g(a, (b)(c, d), f);
```

a and e are indeterminately sequenced (Rule 1)

f and e are indeterminately sequenced (Rule 1)

Function Evaluation



```
void b(...) { e; }
```

```
g(a, (b)(c, d), f);
```

a and e are indeterminately sequenced (Rule 1)

f and e are indeterminately sequenced (Rule 1)

(b) is sequenced before c and d (Rule 2)

Function Evaluation



```
void b(...) { e; }
```

```
g(a, (b)(c, d), f);
```

a and e are indeterminately sequenced (Rule 1)

f and e are indeterminately sequenced (Rule 1)

(b) is sequenced before c and d (Rule 2)

c and d are indeterminately sequenced (Rule 3)

Function Evaluation



```
void b(...) { e; }
```

```
g(a, (b)(c, d), f);
```

a and e are indeterminately sequenced (Rule 1)

f and e are indeterminately sequenced (Rule 1)

(b) is sequenced before c and d (Rule 2)

c and d are indeterminately sequenced (Rule 3)

c and d are sequenced before e (Rule 4)

Function Evaluation



```
void b(...) { e; }
```

```
g(a, (b)(c, d), f);
```

a and e are indeterminately sequenced (Rule 1)

f and e are indeterminately sequenced (Rule 1)

(b) is sequenced before c and d (Rule 2)

c and d are indeterminately sequenced (Rule 3)

c and d are sequenced before e (Rule 4)

a – f are sequenced before the body of g (Rule 2)

Initializer Lists



Each element of a brace initializer is sequenced before the all subsequent elements.

[\[dcl.init\] p17](#)

Construction



```
struct A {  
    A(int i) { cout << i; }  
};
```

```
tuple t0 ( A(0), A(1) ) ;
```

```
tuple t1 { A(0), A(1) } ;
```

Construction



```
struct A {  
    A(int i) { cout << i; }  
};
```

```
tuple t0 ( A(0), A(1) ) ;  
// GCC 8: "10"
```

```
tuple t1 { A(0), A(1) } ;  
// GCC 8: "01"
```


Construction



```
struct A {  
    A(int i) { cout << i; }  
};
```

```
tuple t0 ( A(0), A(1) ) ;  
// GCC 8:  "10"  
// LLVM 7: "01"
```

```
tuple t1 { A(0), A(1) } ;  
// GCC 8:  "01"  
// LLVM 7: "01"
```

Operator Evaluation



The **value computations** (but not the **side effects**) of an operator are sequenced before the **value computations** (but not the **side effects**) of its operands.

[\[intro.execution\] p10 s2](#)

Operator Evaluation



The operand **expressions** to the following operators are **sequenced** left to right:

- $E1 \ \&\& \ E2$ [\[expr.log.and\] p2 s2](#)
- $E1 \ || \ E2$ [\[expr.log.or\] p2 s2](#)
- $E1 \ \ll \ E2$ and $E1 \ \gg \ E2$ [\[expr.shift\] p4](#)
- $E1, \ E2$ [\[expr.comma\] p1 s3](#)
- $E1[E2]$ [\[expr.sub\] p1 s6](#)
- $E1.*E2$ and $E1->*E2$ [\[expr.mptr.oper\] p3 s3](#)

Operator Evaluation



The operands **expressions** to the following operators are **sequenced** right to left:

- $E2 = E1$ and $E2 @ = E1$

[\[expr.ass\] p1 s5](#)

Synchronizes With



Two library operations A and B may be related by the *synchronizes with* relation.

[intro.races] p6

Synchronizes With



Asymmetric: A synchronizes with B does not imply that B synchronizes with A.

Synchronizes With



Ways to achieve **synchronizes with**:

- Atomic acquire/release.
- Mutex lock/unlock.
- Thread create/join.
- Future/promise.
- Parallel algorithm fork/join.

Synchronizes With



```
T data = // ...  
atomic<bool> r(false);
```

```
data = ...  
r.store(1, memory_order_release);
```

Synchronizes with

```
if (r.load(memory_order_acquire)) {  
    T tmp = data;  
    // ...  
}
```

Synchronizes With



```
T data = // ...  
atomic<bool> r(false);
```

```
data = ...  
r.store(1, memory_order_release);
```

Synchronizes with

```
if (r.load(memory_order_acquire)) {  
    T tmp = data;  
    // ...  
}
```

No
synchronizes with

```
data = ...  
r.store(1, memory_order_release);
```

```
if (r.load(memory_order_acquire)) {  
    T tmp = data;  
    // ...  
}
```

Synchronizes With



```
T data = // ...  
std::mutex mtx;
```

Synchronizes
with

```
{ std::lock_guard l(mtx); // Lock  
  T tmp = data;  
  // ...  
} // Unlock
```

```
{ std::lock_guard l(mtx); // Lock  
  T tmp = data;  
  // ...  
} // Unlock
```

```
{ std::lock_guard l(mtx); // Lock  
  T tmp = data;  
  // ...  
} // Unlock
```

Happens Before



Given any two **evaluations** A and B...

If A ***happens before*** B:

Happens Before

Given any two **evaluations** A and B...

If A ***happens before*** B:

- A is **sequenced before** B, or

Happens Before

Given any two **evaluations** A and B...

If A ***happens before*** B:

- A is **sequenced before** B, or
- A **synchronizes with** B, or

Happens Before

Given any two **evaluations** A and B...

If A **happens before** B:

- A is **sequenced before** B, or
- A **synchronizes with** B, or
- For some **evaluation** X, A **happens before** X and X **happens before** B.

[\[intro.races\] p7, p8, p9, p10](#)

Happens Before



Happens before doesn't mean happened before.

Happens Before



```
int x = 0;
```

```
int y = 0;
```

```
void foo()
```

```
{
```

```
    x = y + 1;
```

```
    y = 1;
```

```
}
```

```
// GCC 8.2 -O3 x86-64
```

```
x:
```

```
    .zero 4
```

```
y:
```

```
    .zero 4
```

```
foo():
```

```
    movl y(%rip), %eax
```

```
    movl $1, y(%rip)
```

```
    addl $1, %eax
```

```
    movl %eax, x(%rip)
```

```
    ret
```

Happens Before



Happens before doesn't mean happened before.

Happening before doesn't mean **happens before**.

Happens Before



```
std::atomic<bool> ready = false;
int data = 0;

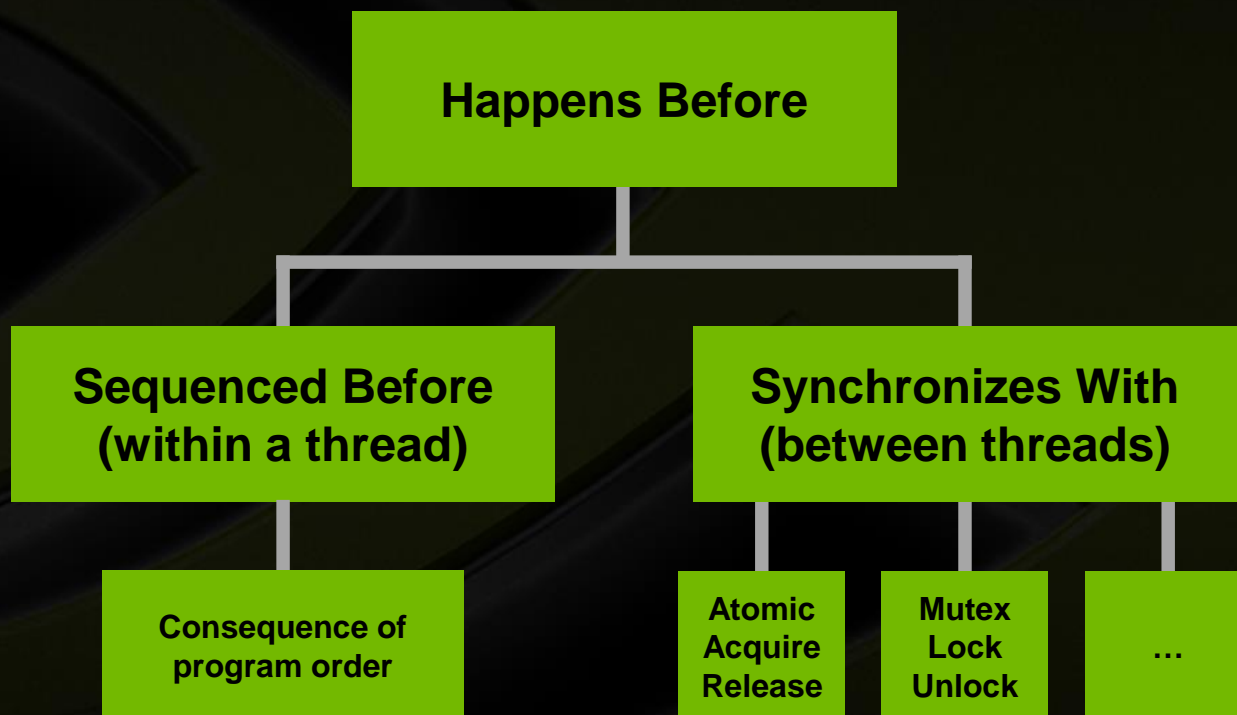
std::thread producer(
    [&] {
        data = 42;
        ready.store(true, memory_order_relaxed);
    });

std::thread consumer(
    [&] {
        if (ready.load(memory_order_relaxed))
            std::cout << data;
    });
```


Happens Before



- **Happens before** describes arbitrary concatenations of **sequenced before** and **synchronizes with**.



Execution Steps



The **evaluations** executed by **threads** are delineated by **execution steps**.

[intro.progress] p3

Execution Steps

The **evaluations** executed by **threads** are delineated by **execution steps**.

[\[intro.progress\] p3](#)

An **execution step** is:

- Termination of the thread.

Execution Steps



The **evaluations** executed by **threads** are delineated by **execution steps**.

[\[intro.progress\] p3](#)

An **execution step** is:

- Termination of the thread.
- An access of a volatile object.

Execution Steps



The **evaluations** executed by **threads** are delineated by **execution steps**.

[\[intro.progress\] p3](#)

An **execution step** is:

- Termination of the thread.
- An access of a volatile object.
- Completion of:
 - A library I/O function call.

Execution Steps



The **evaluations** executed by **threads** are delineated by **execution steps**.

[\[intro.progress\] p3](#)

An **execution step** is:

- Termination of the thread.
- An access of a volatile object.
- Completion of:
 - A library I/O function call.
 - A synchronization operation.

Execution Steps



The **evaluations** executed by **threads** are delineated by **execution steps**.

[\[intro.progress\] p3](#)

An **execution step** is:

- Termination of the thread.
- An access of a volatile object.
- Completion of:
 - A library I/O function call.
 - A synchronization operation.
 - An atomic operation.

Forward Progress



Some atomic operations may **fail spuriously** due to interference from other threads.

Implementations are encouraged, but not required, to prevent spurious failures from indefinitely delaying progress.

[\[intro.progress\] p2.2](#)

Forward Progress



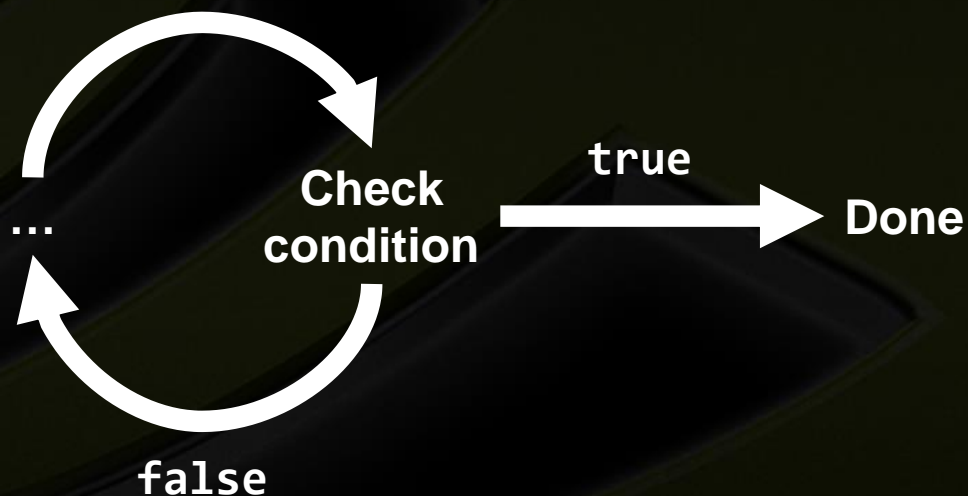
Block: Wait for a **condition** to be **satisfied** before continuing execution.

[defns.block]

Forward Progress



Blocking library functions are considered to continuously execute execution steps while waiting for their condition to be satisfied. Thus, blocking makes progress.



Forward Progress



Forward progress guarantees that something observable should eventually happens.

Execution Steps



Implementations may assume that all threads will eventually perform an execution step.

[\[intro.progress\] p1](#)

Execution Steps



Implementations may assume that all threads will eventually perform an execution step.

[\[intro.progress\] p1](#)

AKA infinite loops that have no observable effects are undefined behavior.

Forward Progress



Three classes of forward progress guarantees:

- Concurrent forward progress.
- Parallel forward progress.
- Weakly parallel forward progress.

Forward Progress



Concurrent forward progress: The thread will make progress, regardless of whether other threads are making progress.

[\[intro.progress\] p7](#)

Forward Progress



Concurrent forward progress example:

- Preemptive OS thread scheduling.
- Unbounded thread pool.

Forward Progress



Parallel forward progress: Once the thread has executed its first execution step, the thread will make progress.

[\[intro.progress\] p9](#)

Forward Progress



Parallel forward progress example:

- **Run-to-completion userspace tasking.**
 - Bounded thread pool.
- **Threads on modern NVIDIA GPUs.**

Forward Progress



Weakly parallel forward progress: The thread is not guaranteed to make progress.

[\[intro.progress\] p11](#)

Forward Progress



Weakly parallel forward progress example:

- Non-preemptive OS thread scheduling.
- Suspendable userspace tasking.
 - Work-stealing task schedulers.
 - Fibers.
- Threadless asynchrony.
 - Lazy execution.
 - C++ coroutines.
- Threads on legacy GPUs.

Forward Progress



Which guarantee does the main thread make?

[\[intro.progress\] p8](#)

Forward Progress



**Which guarantee does the main thread make?
Implementation defined.**

[\[intro.progress\] p8](#)

Forward Progress



Which guarantee do `std::threads` make?

[\[intro.progress\] p8](#)

Forward Progress



**Which guarantee do `std::threads` make?
Implementation defined.**

[\[intro.progress\] p8](#)

Boost Blocking



Boost Blocking: Block on a thread with weaker forward progress while preserving the calling thread's forward progress.

[\[intro.progress\] p14](#)

Boost Blocking



When a thread P **boost blocks** on a set S of other threads, the forward progress guarantees of at least one of the threads in S is temporarily upgraded to P's forward progress guarantee. Repeat until the blocking condition is satisfied.

[\[intro.progress\] p14](#)

Boost Blocking



Boost blocking ensures your children threads make progress, not your siblings.

Boost Blocking



```
struct lazy_thread {  
    function<void()> f;  
  
    void join() {  
        if (f) {  
            // Boost block by running the thread in the  
            // calling thread.  
            f();  
            f = function<void()>{};  
        } else {  
            throw make_error_code(errc::invalid_argument);  
        }  
    }  
};
```

Boost Blocking

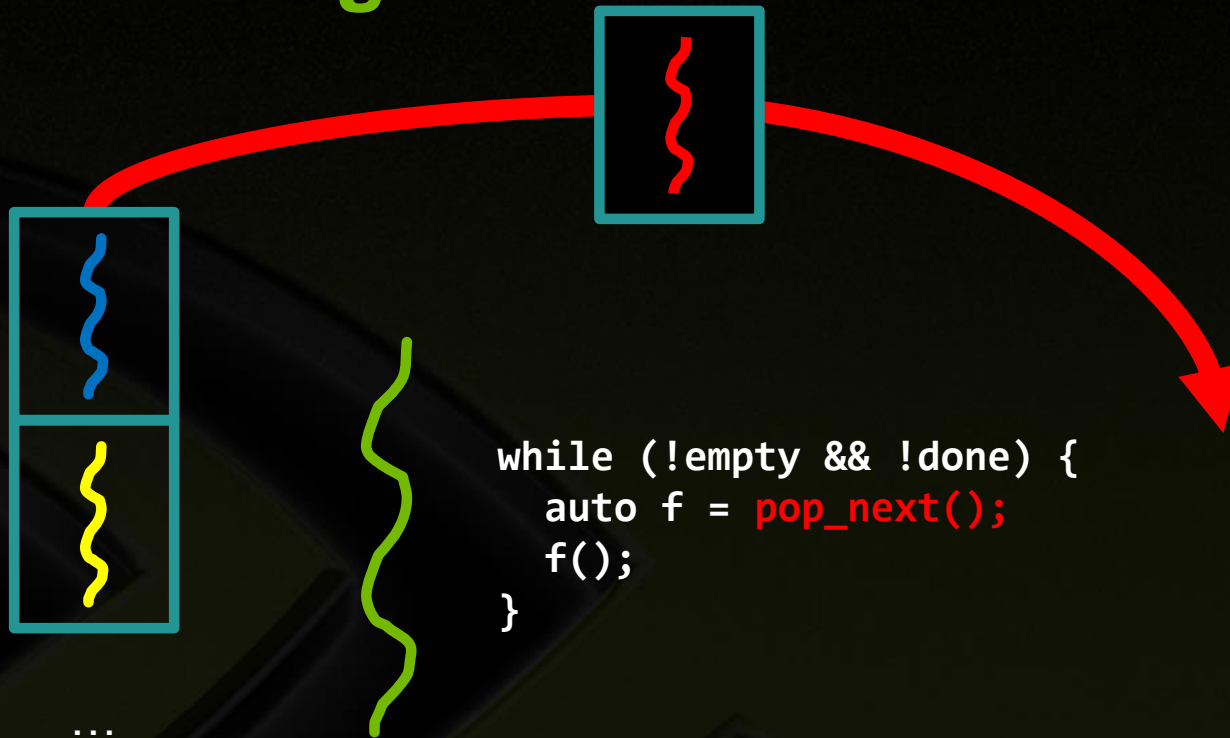


...

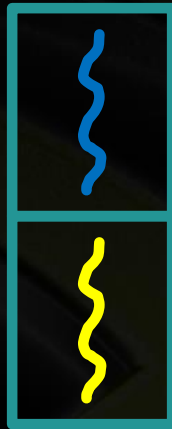


```
while (!empty && !done) {  
    auto f = pop_next();  
    f();  
}
```

Boost Blocking



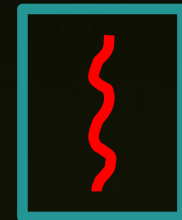
Boost Blocking



...



```
while (!empty && !done) {  
    auto f = pop_next();  
    f();  
}
```



Boost Blocking



```
static_thread_pool pool(...);  
  
auto task = pool.execute_async(  
  
);
```

Boost Blocking



```
static_thread_pool pool(...);  
  
auto task = pool.execute_async(  
    [&] {  
  
    }  
);
```

Boost Blocking



```
static_thread_pool pool(...);

auto task = pool.execute_async(
    [&] {
        auto child0 = pool.execute_async(...);
        auto child1 = pool.execute_async(...);
        ...
        child0.join();
        child1.join();
    }
);
```

Boost Blocking



```
static_thread_pool pool(1);

auto task = pool.execute_async(
    [&] {
        auto child0 = pool.execute_async(...);
        auto child1 = pool.execute_async(...);
        ...
        child0.join();
        child1.join();
    }
);
```

Summary



C++ Execution Model:

- **Threads** evaluate expressions that access and modify flat storage.
- Evaluation within a thread is driven by **sequenced before** relations.
- Interactions between threads is driven by **synchronizes with** relations.
- **Forward progress** promises eventual termination.

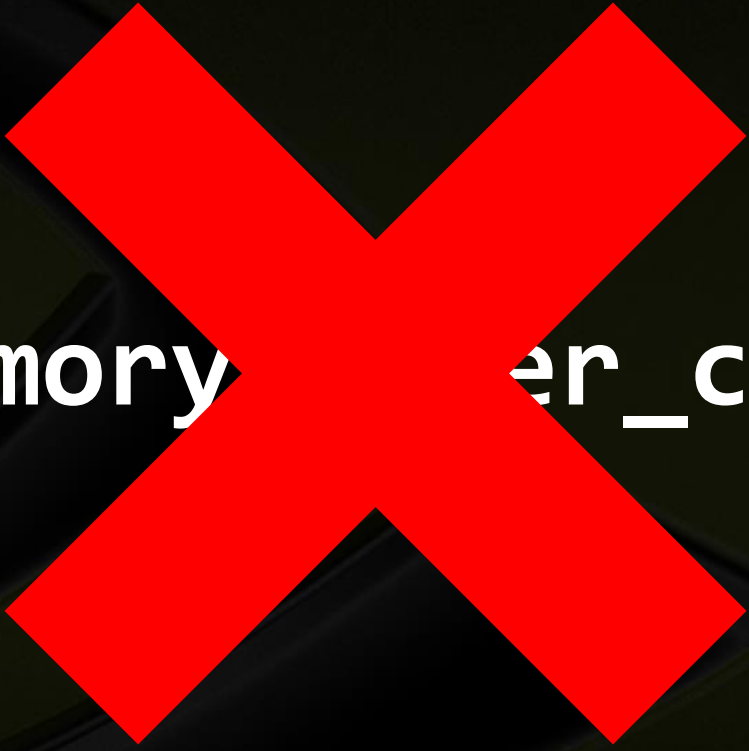
C++17 and beyond

std::memory_order_consume

Caveats



`std::memory_order_consume`

A large, bright red 'X' mark is superimposed over the text, indicating that the feature is deprecated or not recommended.

Summary



C++ Execution Model:

- **Threads** evaluate expressions that access and modify flat storage.
- Evaluation within a thread is driven by **sequenced before** relations.
- Interactions between threads is driven by **synchronizes with** relations.
- **Forward progress** promises eventual termination.



@blelbach