C++ Memory Model

C++ User Group Berlin 17/12/2013

Valentin Ziegler

Fabio Fracassi

think-cell
Software GmbH

What this talk is not

- about why we need to do multithreaded programming
- about how to do multithreaded programming
- about how to use locks, how to prevent common problems with them
- about how not to have deadlocks or livelocks
- about how to do lockfree programming

… it is about what we need to reason about concurrent code

you might still learn something about multithreaded programming …

… maybe just that it is even more complicated than you thought it was …

Your computer does not execute the program you wrote

Before we start

- Computers do not execute the program you write
  - the compiler will optimize your program
    - loop fusion, …
  - the CPU will optimize your instructions
    - branch-prediction, …
  - the cache will optimize your loads and stores
    - prefetching, …
- They will execute a program that will behave as-if it was yours
- Can’t get any kind of performance without that
- We will never know which changes the system made because we can not observe them.

For us all of these optimizations look like the system **reordered** the memory accesses

Before we start

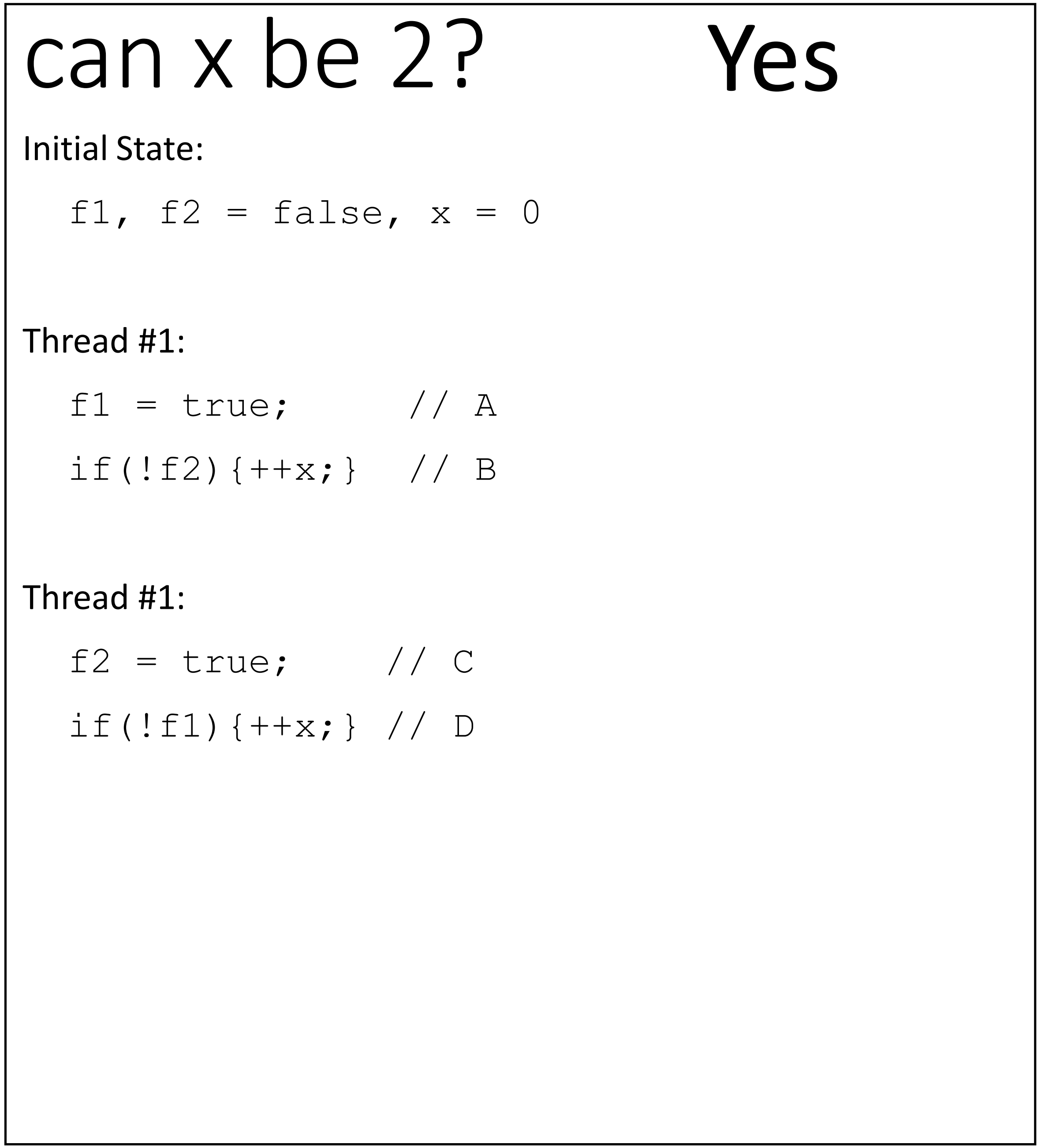
- So what happens when a second thread comes in?
  - Now the ordering of memory access becomes observable
  - At least for the data that is/might be shared

- Two options:
  - System stops doing the optimizations that have become observable
  - We cope with the unpredictably ordered memory accesses

$\Rightarrow$ MEH!

Can we cope?

- No!



can x be 2? Yes

Initial State:

```
  f1, f2 = false, x = 0
```

Thread #1:

```
  f1 = true;       // A
  if(!f2){++x;}    // B
```

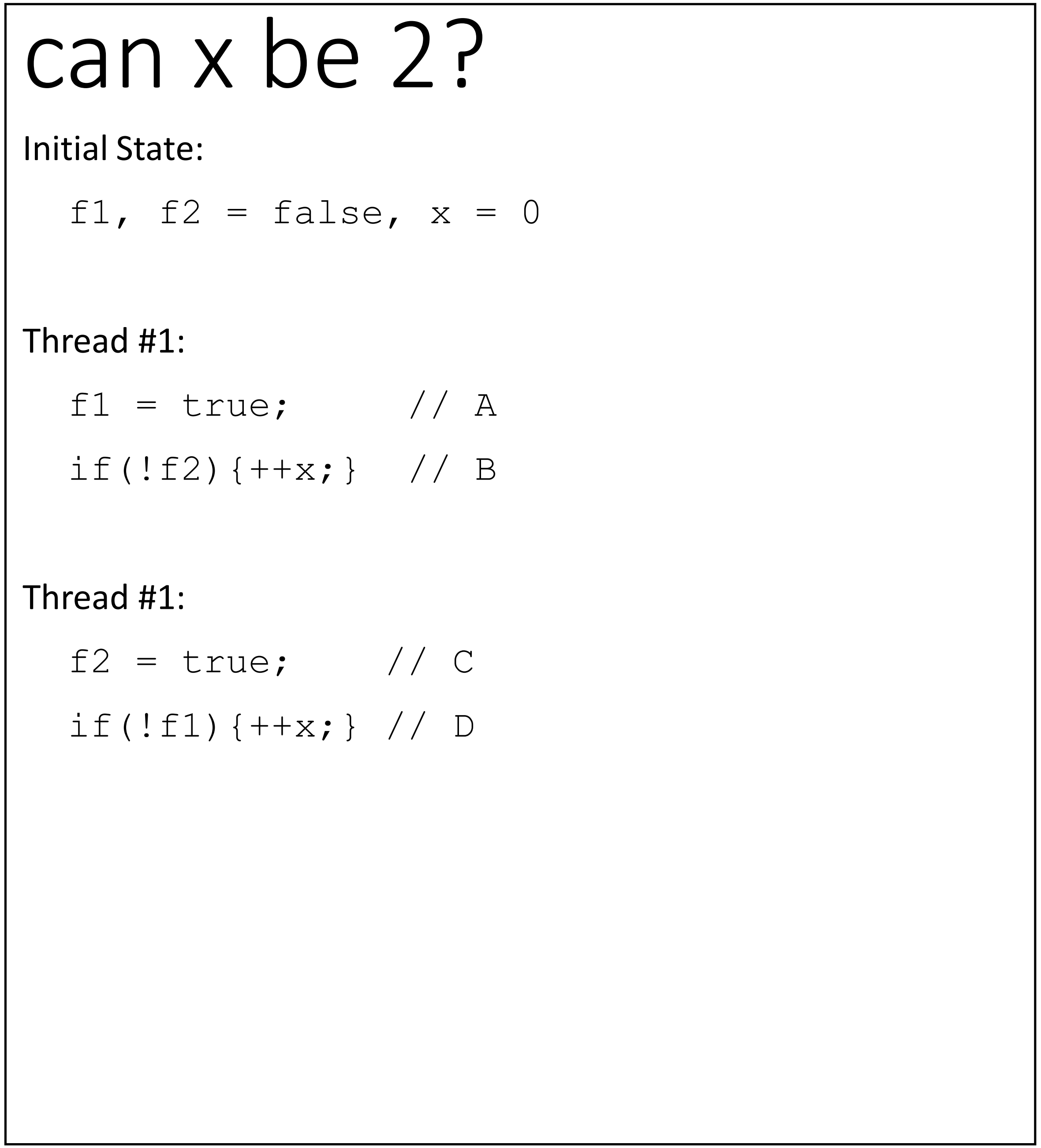
Thread #1:

```
  f2 = true;      // C
  if(!f1){++x;} // D
```

Can we cope?

- No!
- No, really not!
- We cannot implement critical sections without consistently ordered memory access
- We’d lose causality!

and we really don’t want causality to go all *wibbily wobbly timey wimey* on us ... debugging is hard enough in a world of *strict progression of cause to affect*



can x be 2?

Initial State:

```
f1, f2 = false, x = 0
```

Thread #1:

```
f1 = true;        // A
if(!f2){++x;}   // B
```

Thread #1:

```
f2 = true;      // C
if(!f1){++x;} // D
```

But we did concurrent programing, before

- We manually used platform specific primitives to synchronize our memory accesses
  - Hardware provided special instructions to flush caches or synchronize memory accesses
  - Compilers either used special build-in primitives or were taught to recognize these instructions, to prevent broken optimizations

Memory Model

- describes the interactions of threads through memory and their shared use of data.
- allow the system to make optimizations to your program without breaking it.

fairly new concept. Java has a formalized memory model since 2005, C++ since 2011

not to be confused with **memory addressing models**, which have largely gone the way of the Dodo.

a few boring definitions …

Data Race

conflicting action [intro.multithread(1.10)/4] (sometimes known as race condition)

two (or more) actions that access the same *memory location* and at least one of them is a write

memory location [intro.memory(1.7)/3]

an object of scalar type or a maximal sequence of adjacent non-zero width bit-fields

data race [intro.multithread(1.10)/21] (sometimes known as race condition)

two *conflicting actions* in different threads and neither *happens before* the other.

Data Race

conflicting action [intro.multithread(ondition)

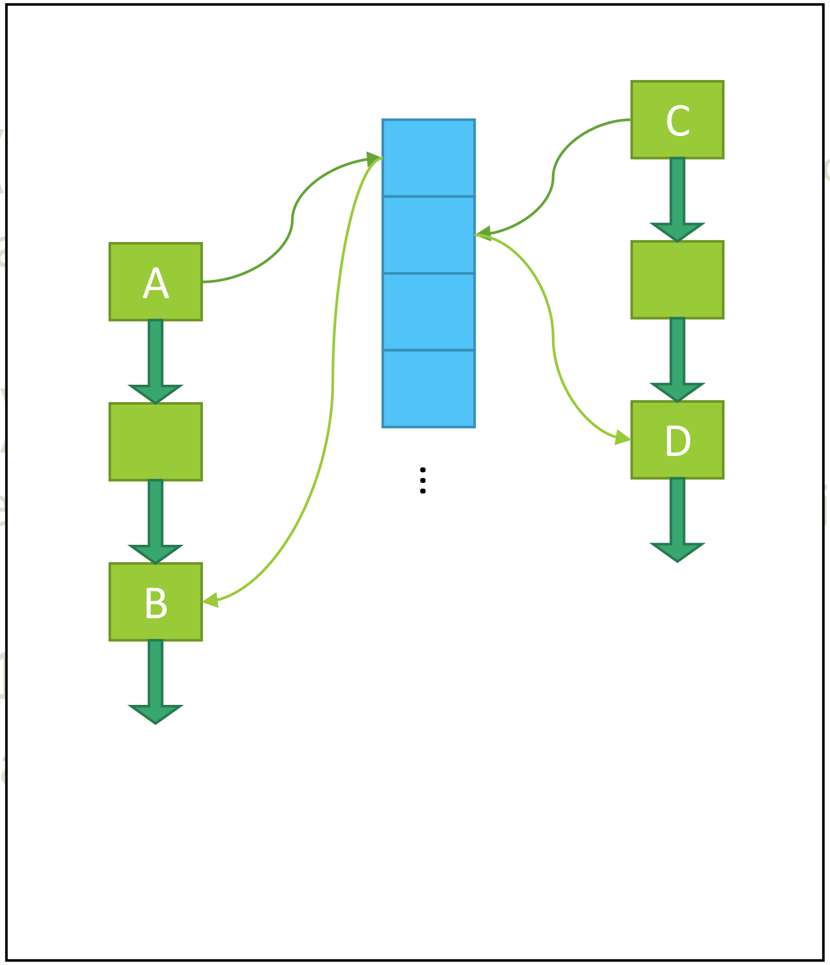
two (or more) actions that access the sa st one of them is a write

memory location [intro.memory(1.7)

an object of scalar type or a maximal se idth bit-fields

data race [intro.multithread(1.10)/21

two *conflicting actions* in different thre the other.



Data Race

conflicting action [intro.multithread(ondition)

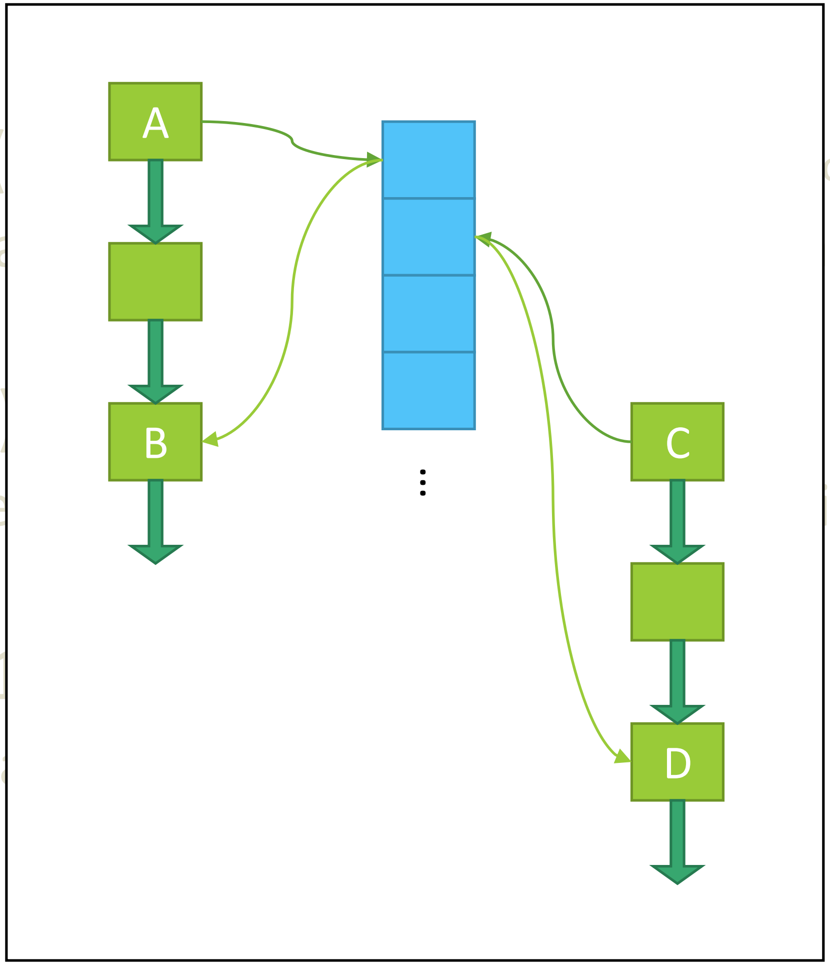
two (or more) actions that access the sa st one of them is a write

memory location [intro.memory(1.7)

an object of scalar type or a maximal se idth bit-fields

data race [intro.multithread(1.10)/21

two *conflicting actions* in different threa the other.



Data Race

conflicting action [intro.multithread(ondition)

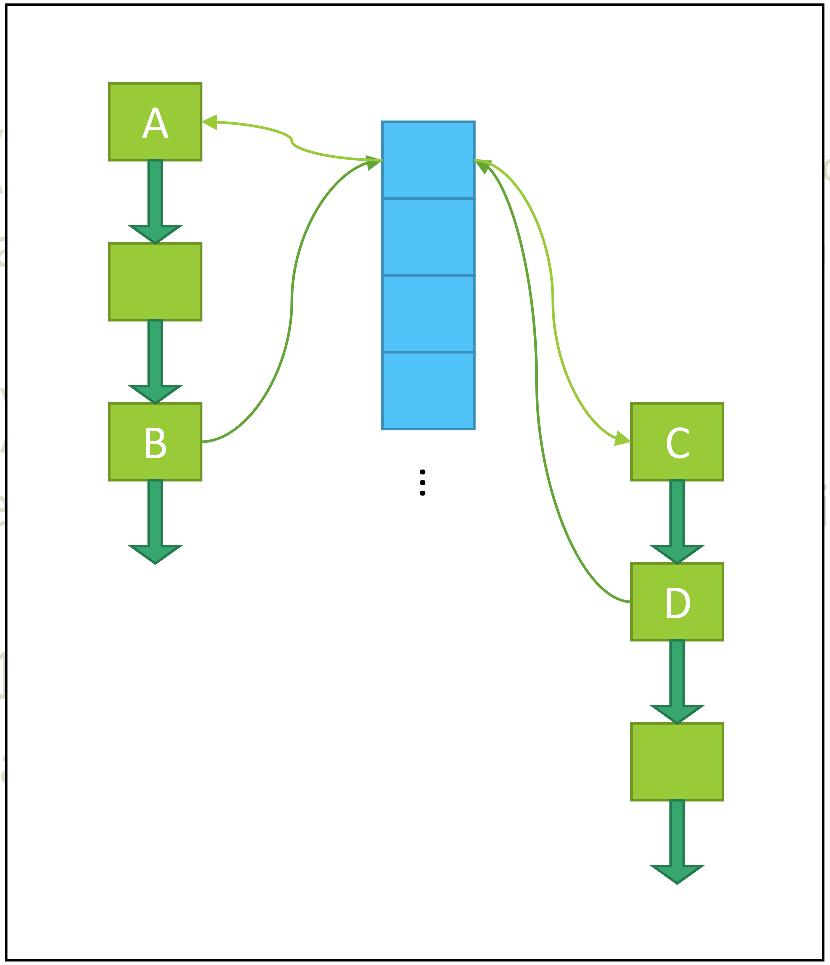
two (or more) actions that access the sa st one of them is a write

memory location [intro.memory(1.7)

an object of scalar type or a maximal se idth bit-fields

data race [intro.multithread(1.10)/21

two *conflicting actions* in different threa the other.



Data Race

conflicting action [intro.multithread(ondition)

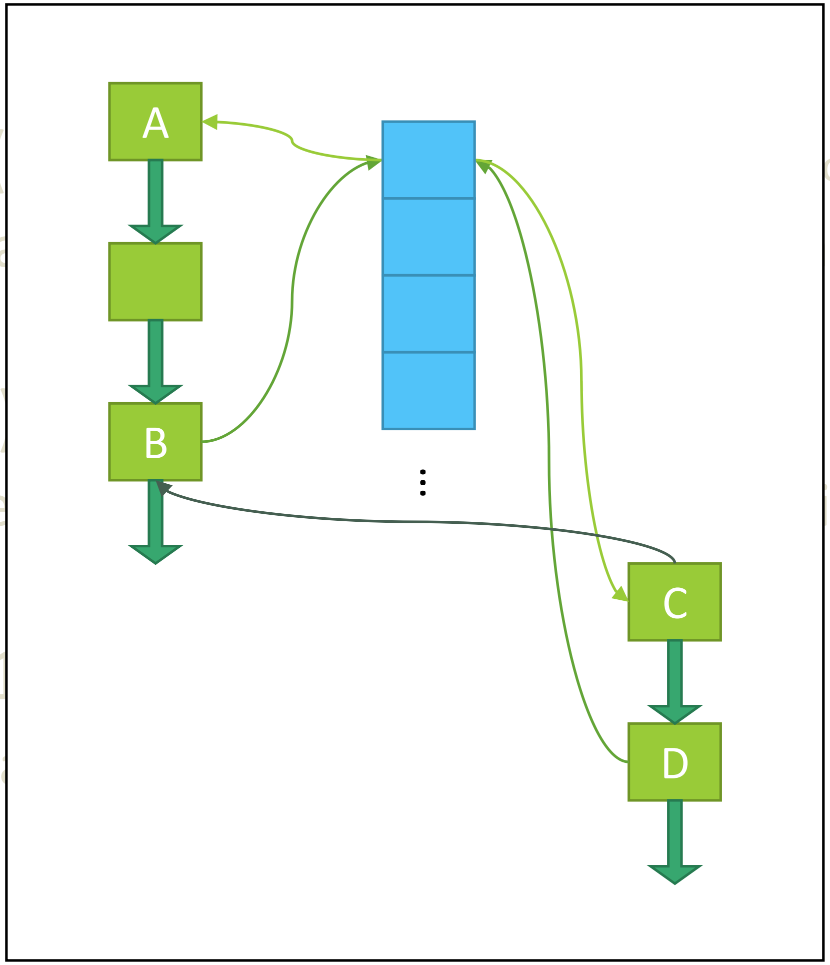
two (or more) actions that access the sa st one of them is a write

memory location [intro.memory(1.7)

an object of scalar type or a maximal se idth bit-fields

data race [intro.multithread(1.10)/21

two *conflicting actions* in different threa the other.



Data Race

conflicting action [intro.multithread(1.10)/4] (sometimes known as race condition)

two (or more) actions that access the same *memory location* and at least one of them is a write

memory location [intro.memory(1.7)/3]

an object of scalar type or a maximal sequence of adjacent non-zero width bit-fields

data race [intro.multithread(1.10)/21] (sometimes known as race condition)

two *conflicting actions* in different threads and neither *happens before* the other.

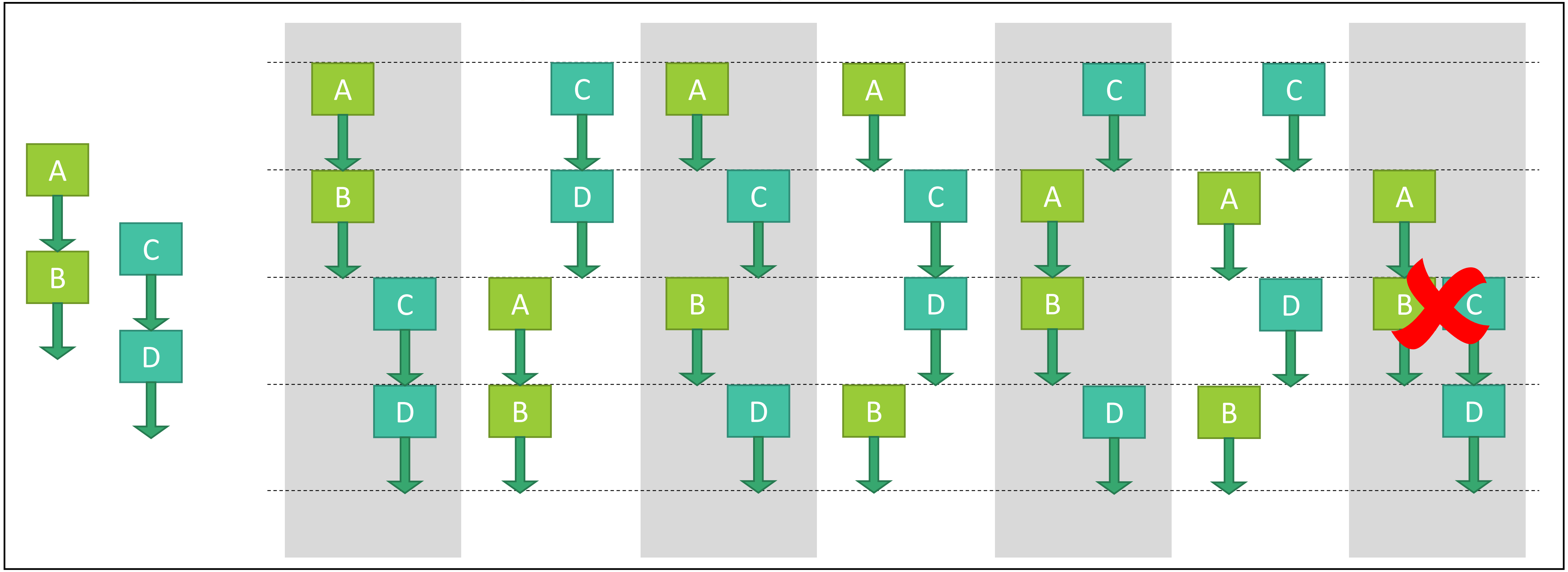
Sequential Consistency

sequential consistency [Leslie Lamport, 1979]

> the result of any execution is the same as-if the operations of all threads are executed in some sequential order, and the operations of each thread appear in this sequence in the order specified by their program

Sequential Consistency

sequential consistency [Leslie Lamport, 1979]



The C++ memory model

sequential consistency for data-race-free programs
SC-DRF

Here is the deal:

- We do not write data races into our program
- The system guarantees sequentially consistent execution

So how do we prevent data races?

- Do not share our data!
- Synchronize our data access

synchronize (the easy way)…

Locks

lock shared memory location for exclusive access while in use

+ leaves intra-thread optimization alone
    - but what happens in the critical section stays in the critical section
    - => critical sections prevent memory access reordering across them

+ synchronizes with other threads

+ it “just works” …

- requires care on **every use** of a memory location

- prone to races, deadlocks and livelocks

Locks

Good time

```
int fun_money = atm.get(limit);
{  auto in_lasVegas = std::lock_guard<std::mutex>(lasVegas);
   fun_money = gamble(fun_money);
}
socialNet.post(“Had fun in Vegas, won $” + (fun_money - limit));
```

Locks

Good time?

```
int fun_money = atm.get(limit);
                                  // lost all my money before even
fun_money = gamble(fun_money);    // getting to vegas ☹
{  auto in_lasVegas = std::lock_guard<std::mutex>(lasVegas);
}
socialNet.post(“Had fun in Vegas, won $” + (fun_money - limit));
```

Locks

Good time?

```
int fun_money = atm.get(limit);
{  auto in_lasVegas = std::lock_guard<std::mutex>(lasVegas);
}
fun_money = gamble(fun_money); // got jailed for illegal gambling ☹
socialNet.post(“Had fun in Vegas, won $” + (fun_money - limit));
```

Locks

Good time – as long as we respect the borders

```
int fun_money = atm.get(limit);
{  auto in_lasVegas = std::lock_guard<std::mutex>(lasVegas);
   fun_money = gamble(fun_money);
}
socialNet.post(“Had fun in Vegas, won $” + (fun_money - limit));
```

Locks

Good time – as long as we respect the borders

```
int fun_money = atm.get(limit);
{  lasVegas.lock(); // entering Las Vegas
   fun_money = gamble(fun_money);
   lasVegas.unlock(); // leaving Las Vegas
}
socialNet.post(“Had fun in Vegas, won $” + (fun_money - limit));
```

Locks

Good time – as long as we respect the borders

```
int fun_money = atm.get(limit);
{  lasVegas.lock(); // entering Las Vegas – no reordering allowed!
   fun_money = gamble(fun_money);
   lasVegas.unlock(); // leaving Las Vegas – no reordering allowed!
}
socialNet.post(“Had fun in Vegas, won $” + (fun_money - limit));
```

Locks

Good time?

```
{  lasVegas.lock(); // entering Las Vegas
   int fun_money = atm.get(limit);
   fun_money = gamble(fun_money);
   socialNet.post(“Had fun in Vegas, won $” + (fun_money - limit));
   lasVegas.unlock(); // leaving Las Vegas
}
```

Locks

Good time – sure no problem

```
{  lasVegas.lock(); // entering Las Vegas – no gambling before this
   int fun_money = atm.get(limit);
   fun_money = gamble(fun_money);
   socialNet.post(“Had fun in Vegas, won $” + (fun_money - limit));
   lasVegas.unlock(); // leaving Las Vegas – no gambling after this
}
```

Locks and barriers

- locks imply barriers
  - full barriers would be too restrictive
  - acquire on locking / release on unlocking is sufficient
- the C++ threading library provides locks with the appropriate acquire/release semantics
- if you use locks to correctly protect your shared memory locations the system guarantees sequentially consistent execution.
  - What is strange about the previous example?

Lockfree data structures

try to update a shared memory location and retry if someone else interfered

- based on `std::atomic<>`
  - needs hardware support
  - not all platforms provide lockfree atomics

+ tag the shared variable not every place it is used

- harder than it looks
  - lockfree data structures are still a frontier in research

Don’t do this at Work!

Lockfree data structures

- if you us
  shared r
  consiste

operations on atomics are indivisible

```
template<typename T> class stack {
   struct node{T data; node* next; node(T const& data_):data(data_){}};
   std::atomic<node*> head;
public:
   void push(T const& data) {
      node* const newNode = new node(data);
      newNode->next = head.load(); // equiv: …->next = head;
      while(!head.compare_exchange_weak(newNode->next, newNode))
         ;                                      ↑            ↑
   }                                        Expected      Desired
};
```

Lockfree data structures

try to update a shared memory locat se
interfered

- based on `std::atomic<>`
  - needs hardware support
  - not all platforms provide lockfree ato

+ tag the shared variable not every pl

- harder than it looks

  - lockfree data structures are still a fror

These are not the `volatiles`
you are looking for!

- in C++ the concept is spelled `std::atomic<>`
- `volatile` is for “talking” to stuff that lives outside the memory model (e.g. Hardware Registers)
- provides even **fewer** guarantees than atomics
- does **not** provide inter-thread synchronization
- it is “just like IO”

Are we there yet?

- If you stay in this world you are fine
  - as long as you apply locks correctly
  - and/or as long as you implement your lockfree data structures correctly
- SC-DRF is the default C++ memory model
  - also the (only) memory model of Java and C#
- On modern hardware you will almost always get nearly optimal performance

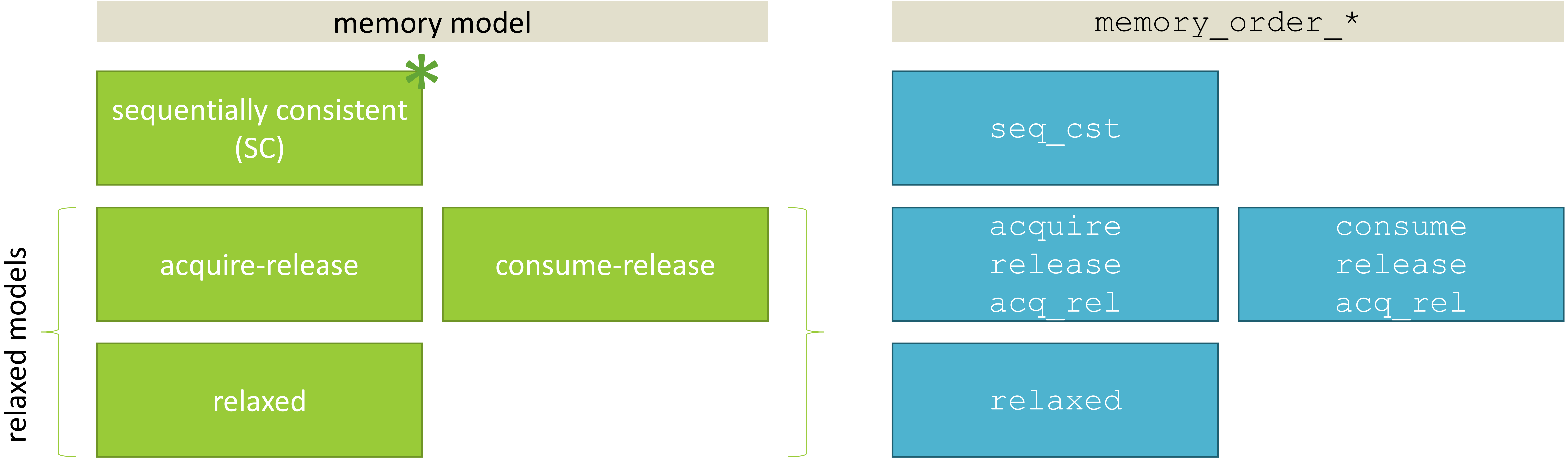
No, and there is still a long way to go

- This wouldn’t be C++ if we couldn’t make it a bit more complex
  - to tell the system that we still know its job better than it does
  - so that we can squeeze the last ounce of performance out of it

- Some of us are just not happy if we cannot twiddle all the knobs

Memory Order

Why stop at one memory model when we can have 3 (and a half)?



memory model

sequentially consistent (SC) *

acquire-release

consume-release

relaxed

relaxed models

memory_order_*

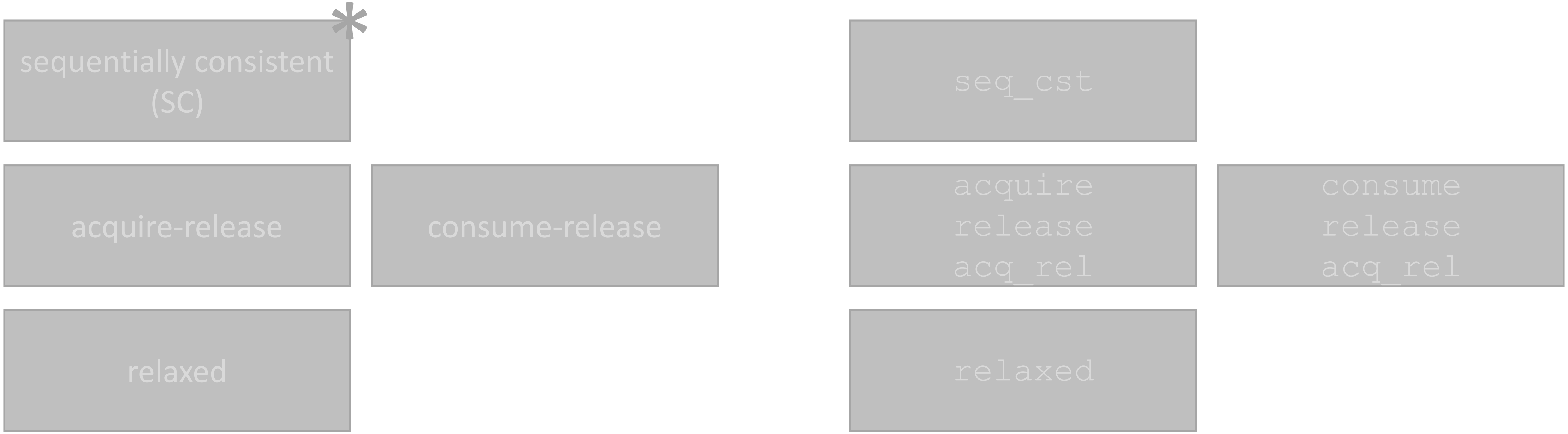
seq_cst

acquire
release
acq_rel

consume
release
acq_rel

relaxed

down to the bottom – no memory model



sequentially consistent (SC)*

acquire-release

consume-release

relaxed

seq_cst

acquire
release
acq_rel

consume
release
acq_rel

relaxed

safe?

Initial State:

```
int c = 0
```

Thread #1,#2, ...:

```
for (int i=0;i<100;++i) {
  …
  ++c;
  …
}
```

Thread main:

```
start_n_threads();
join_n_threads();
assert(100*n == c);✘
```

No, classical data race

Because the system may implement `++c` as:

```
for (int i=0;i<100;++i) {
  …
  { // ++c;
    register int tmp = c;
    tmp = tmp + 1;
    c = tmp;
  }
  …
}
```

Enter `std::atomic<>` for basic guarantees

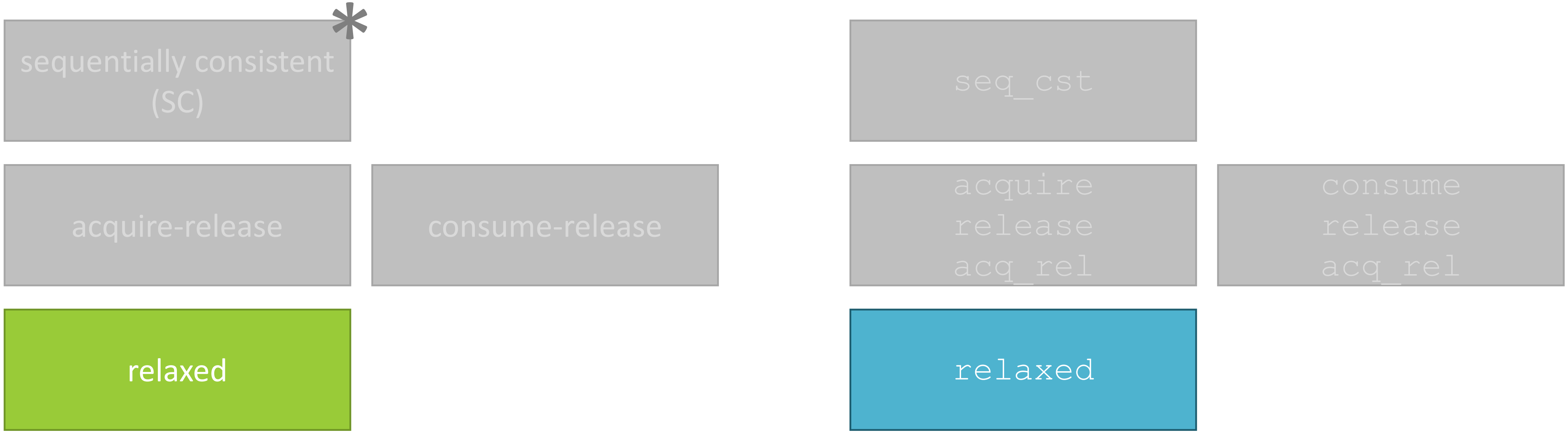
- atomics guarantee that loads and stores are done atomically
  - think: `std::atomic<BigBigInt> = 9’000’000’000’000’000’000’000’000;`
- provide facilities to atomically implement **R**ead-**M**odify-**W**rite operations

```
oldval = c;
while(!c.compare_exchange_weak(oldval, oldval+1))
  ;
```

↑ Expected ↑ Desired

- provide common RMW operations:
  - increment, logic operations, `fetch_add`/`fetch_sub`

The relaxed model



sequentially consistent (SC) *

acquire-release

consume-release

relaxed

seq_cst

acquire
release
acq_rel

consume
release
acq_rel

relaxed

The relaxed model guarantees scarcely anything

- operations on the **same memory location** in the **same thread** will not be reordered
- once a thread has seen a value subsequent reads on the same thread cannot see an earlier value

safe? Yes

Initial State:

```
atomic<int> c = 0
```

Thread #1,#2, …:

```
for (int i=0;i<100;++i) {
  …
  c.fetch_add(1, memory_order_relaxed);
  …
}
```

Thread main:

```
start_n_threads();
join_n_threads();
assert(100*n == c);✓
```

progress guaranteed? Yes

Initial State:

```
atomic<int> c = 0
```

Thread #1,#2, …:

```
for (int i=0;i<100;++i) {
  …
  c.fetch_add(1, memory_order_relaxed);
  …
}
```

Thread main:

```
int old_c = c;
start_n_threads();
  int c_now = c.load(memory_order_relaxed);
  assert(old_c <= c_now);✓
  old_c = c_now;
join_n_threads();
assert(100*n == c);✓
```

0 or 1?

cannot be sure!

Initial State:

```
  f1, f2 = false, x = 0
```

Thread #1:

```
  f1.store(true, memory_order_relaxed); //A
  f2.store(true, memory_order_relaxed); //B
```

Thread #2:

```
  while(!f2.load(memory_order_relaxed)); //C
  if(f1.load(memory_order_relaxed)){++x;}//D
```

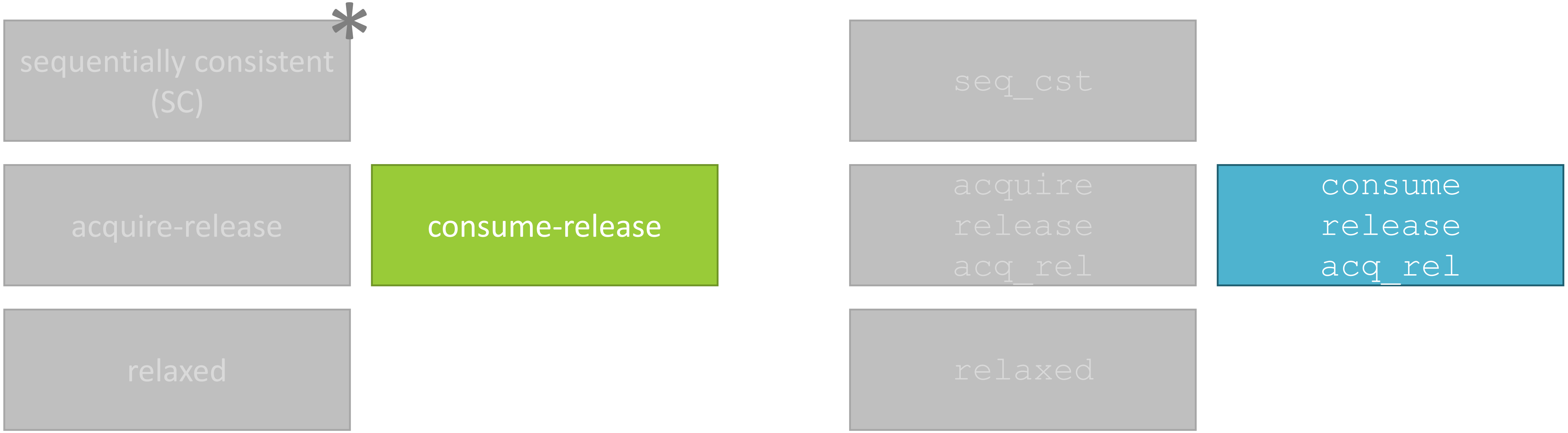
The relaxed model guarantees scarcely anything

- operations on the **same memory location** in the **same thread** will not be reordered
- once a thread has seen a value subsequent reads on the same thread cannot see an earlier value
- No (automatic) inter-thread synchronization!
  - needs to be done manually with fences(aka barriers)
  - `std::atomic_thread_fence(memory_order)`
  - manual fences are fairly expensive, they force all memory operations over all threads to synchronize

A note of caution:

“don’t fall into the trap of thinking that *synchronize* is a relationship between statements in your source code. It isn’t! It’s a relationship between operations which occur at runtime, based on those statements”

The consume/release model



sequentially consistent (SC)*

acquire-release

consume-release

relaxed

seq_cst

acquire release acq_rel

consume release acq_rel

relaxed

What does consume mean?

- a ***read-consume*** *operation R* is correctly paired with a **write-release** operation *W*.
- All Operations in the releasing thread <u>preceding</u> the write-release **inter-thread-happen-before** an operation *X* in the acquiring thread, if *R* ***carries-a-dependency-to*** *X*.

Typical examples for R *carries-a-dependency-to* X:

- X dereferences a pointer obtained by R
- X is accessing array at index obtained R

who has the answer?

Setup:

```
  struct X { int i; }
  int c;
  std::atomic<X*> px;
```

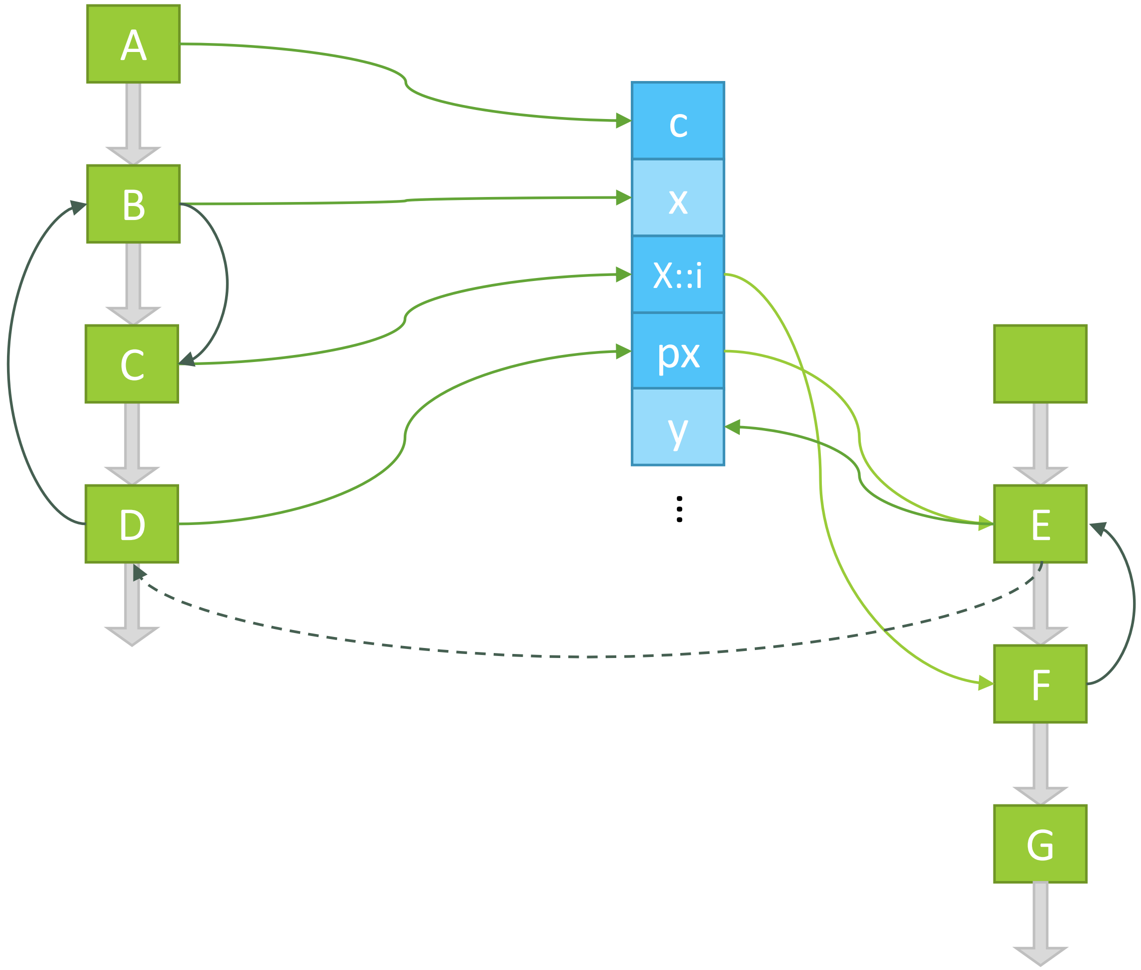
Thread #1:

```
  c = 42;                              //A
  auto x = new X;                      //B
  x->i = 42;                           //C
  px.store(x, memory_order_release);   //D
```

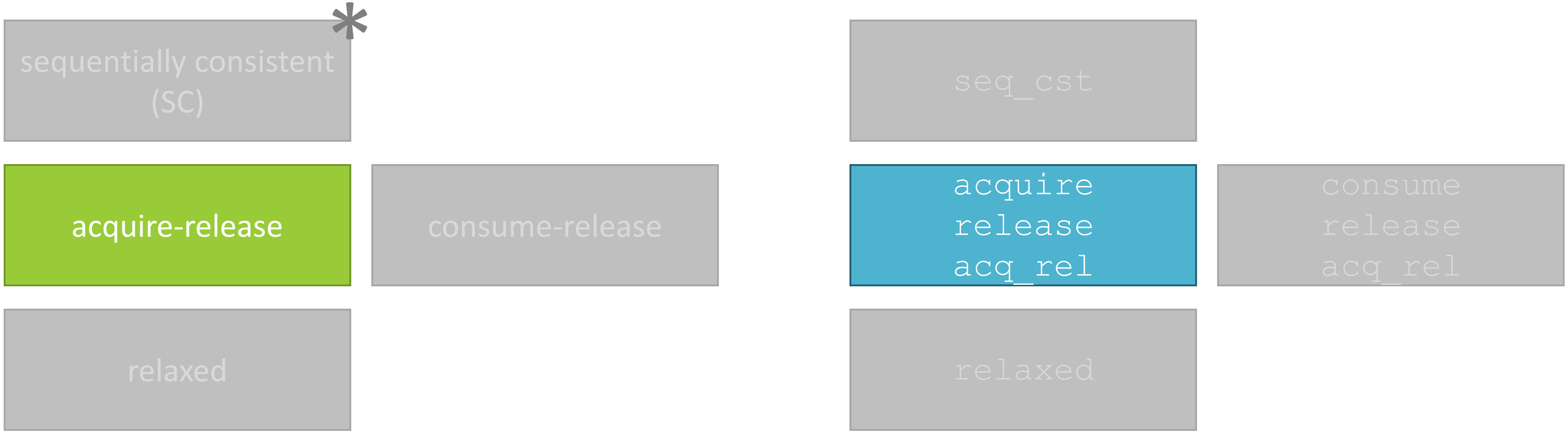
Thread #2:

```
  X* y;                             // ⬃ E
  while(!y=px.load(memory_order_consume));
  assert(42 == y->i);✓                 //F
  assert(42 == c);✗                    //G
```

`y->i` does



The acquire/release model



sequentially consistent (SC) *

acquire-release

consume-release

relaxed

seq_cst

acquire
release
acq_rel

consume
release
acq_rel

relaxed

What does acquire/release mean

- a **read-acquire** operation that is **correctly paired** with a **write-release** operation **introduces synchronization** between those two threads

- All Operations in the releasing thread preceding the write-release **inter-thread-happen-before** all operations following the read-acquire in the acquiring thread.

- **R**ead-**M**odify-**W**rite operations can have acquire, release or both(`acq_rel`) semantics

who has the answer?

Setup:

```
  struct X { int i; }
  int c;
  std::atomic<X*> px;
```

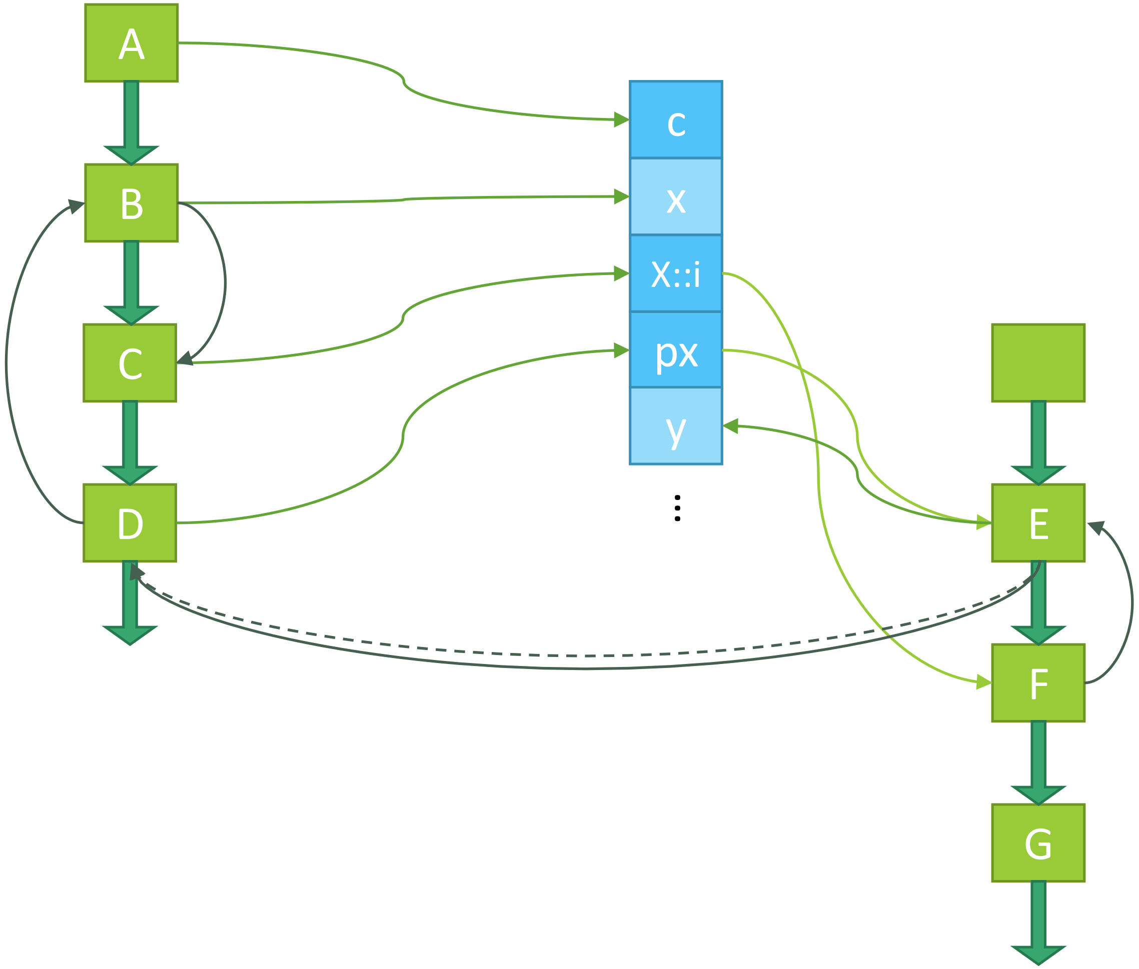
Thread #1:

```
  c = 42;                                   //A
  auto x = new X;                           //B
  x->i = 42;                                //C
  px.store(x, memory_order_release);        //D
```

Thread #2:

```
  X* y;                                 // ⇙ E
  while(!y=px.load(memory_order_acquire));
  assert(42 == y->i);✓                      //F
  assert(42 == c);✓                         //G
```

`y->i` does and so does `c`



is x the answer? Yes

Initial State:

```
  f1, f2 = false, x = 0
```

Thread #1:

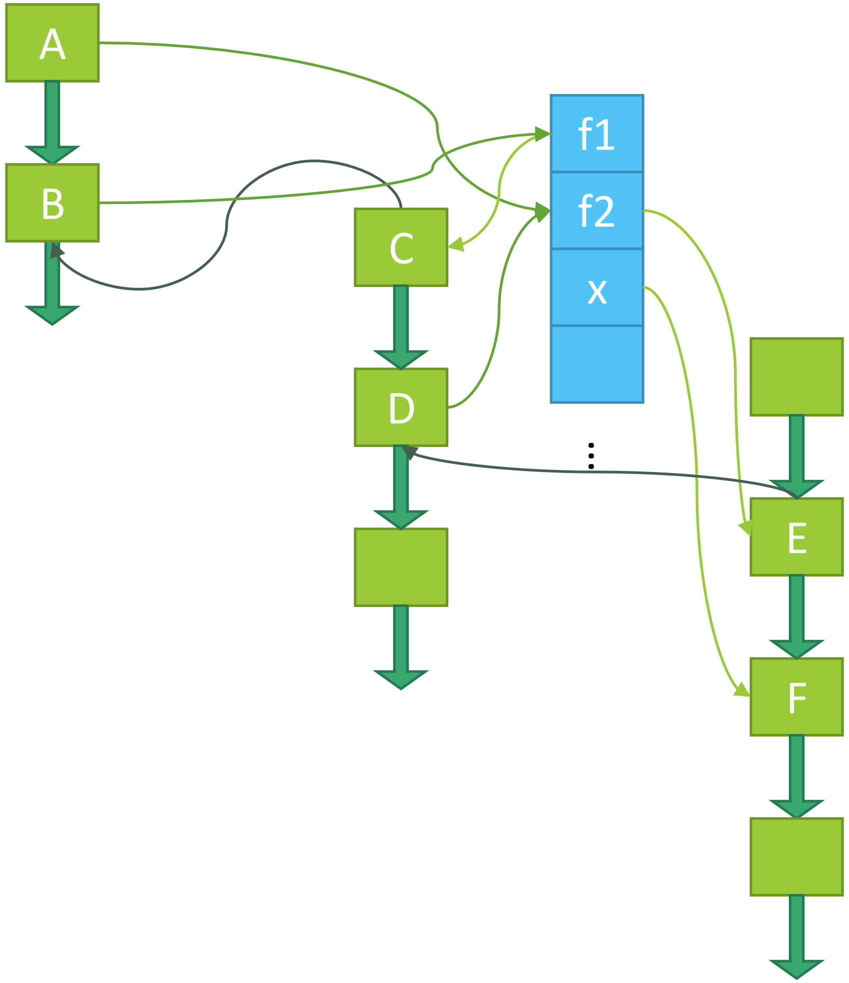
```
  x = 42;                                //A
  f1.store(true, memory_order_release);  //B
```

Thread #2:

```
  while(!f1.load(memory_order_acquire));//C
  f2.store(true, memory_order_release);  //D
```

Thread #3:

```
  while(!f2.load(memory_order_acquire));//E
  assert(42 == x);✓                      //F
```



value of x? 0 , 1 or 2

Initial State:

```
  f1, f2 = false, x = 0
```

Thread #1:

```
  f1.store(true, memory_order_release);//A
```

Thread #2:

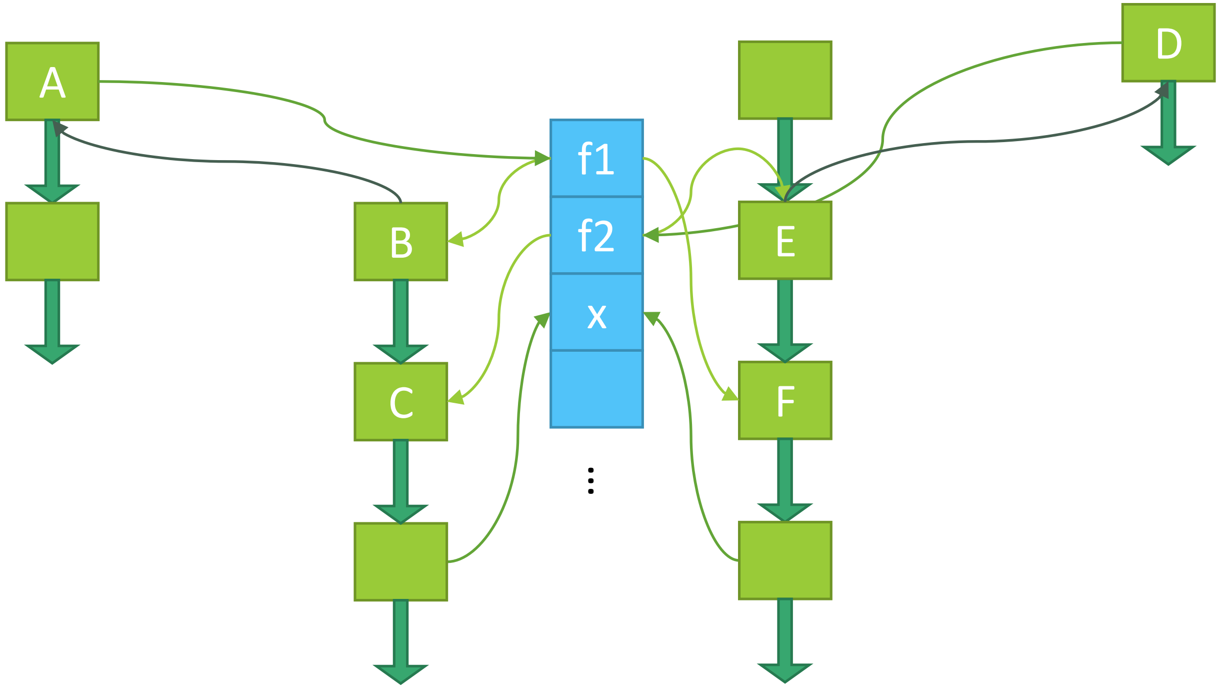
```
  while(!f1.load(memory_order_acquire));  //B
  if(f2.load(memory_order_acquire)){++x;} //C
```

Thread #3:

```
  while(!f2.load(memory_order_acquire));  //E
  if(f1.load(memory_order_acquire)){++x;} //F
```

Thread #4:

```
  f2.store(true, memory_order_release);//D
```



A < B < C

D < E < F

x = 2? D < C && A < F

x = 1? D > C XOR A > F

x = 0? D > C && A > F

Wait, what?
A variable can ever have more than one value?

it is just a memory location, a bunch of bits at a specific location in memory

- quick Q: how many MB of Cache do you have?
- L2-Cache?
- how many L3-Caches do you have?
- How do those interact?
- What if cores share data?

the pointer is a lie!

value of x?

Initial State:

```
  f1, f2 = false, x = 0
```

Thread #1:

```
  f1.store(true, memory_order_release);//A
```

Thread #2:

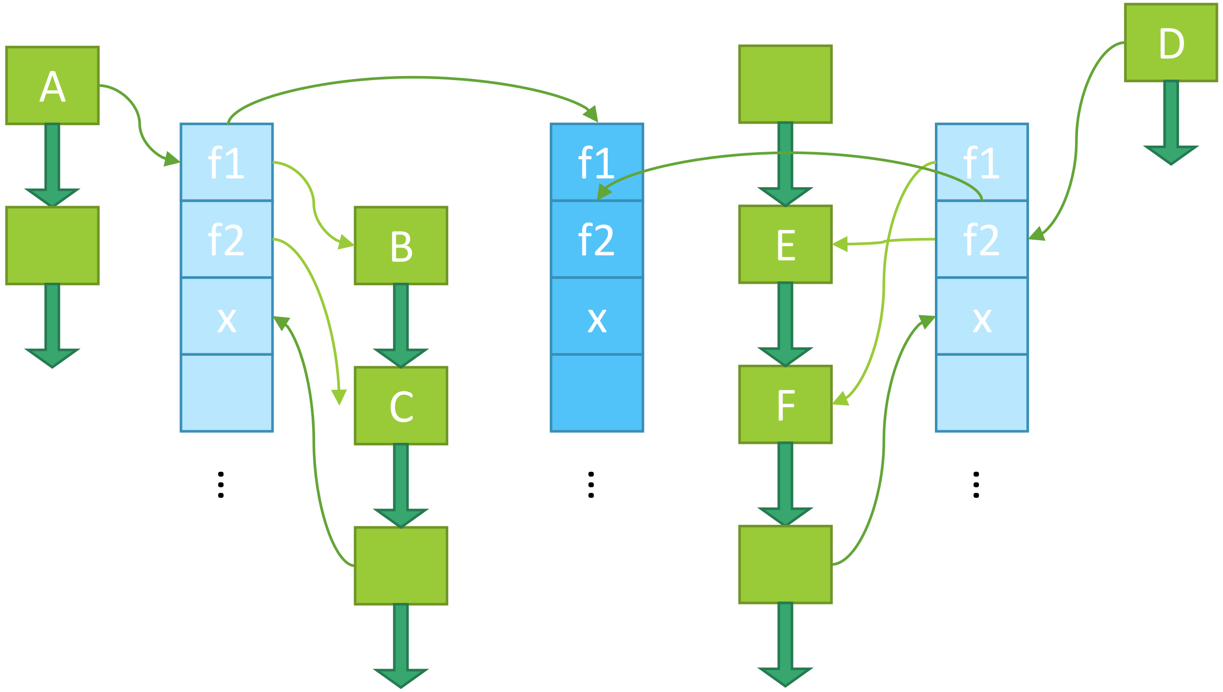
```
 while(!f1.load(memory_order_acquire));  //B
 if(f2.load(memory_order_acquire)){++x;} //C
```

Thread #3:

```
 while(!f2.load(memory_order_acquire));  //E
 if(f1.load(memory_order_acquire)){++x;} //F
```

Thread #4:

```
  f2.store(true, memory_order_release);//D
```



value of x?

Initial State:

```
f1, f2 = false, x = 0
```

Thread #1:

```
f1.store(true, memory_order_release);//A
```

Thread #2:

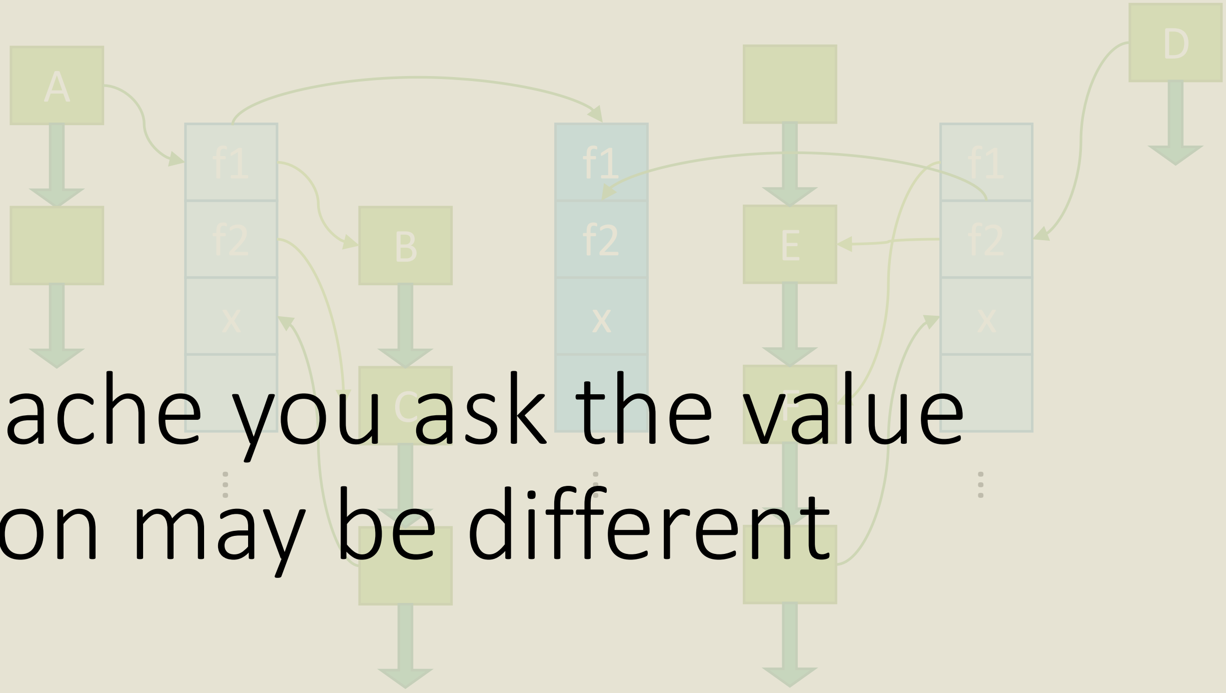
```
while(!f1.load(memory_order_acquire));  //B
if(f2.load(memory_order_acquire)){++x;} //C
```

Thread #3:

```
while(!f2.load(memory_order_acquire));  //E
if(f1.load(memory_order_acquire)){++x;} //F
```

Thread #4:

```
f2.store(true, memory_order_release);//D
```



Depending which cache you ask the value of a memory location may be different

Why is that

- the acquire/release model does not guarantee that a store to an atomic value becomes visible to all threads at the same time
- on some systems skipping this value propagation can have a positive performance impact

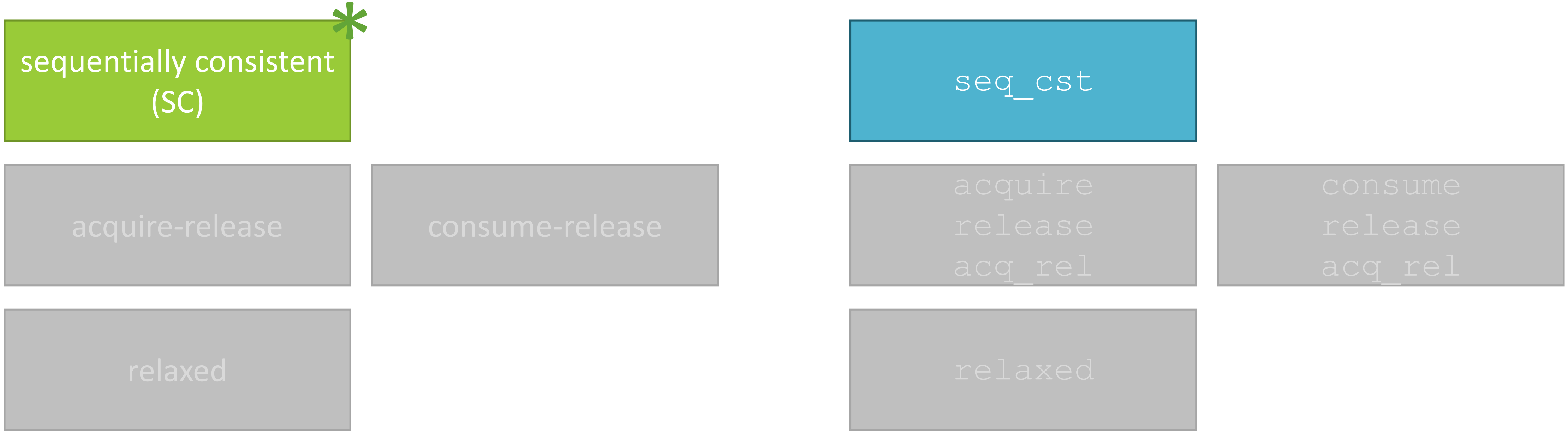
Legitimate use-cases for the relaxed models?

- target platform is ARM (<v8) or PowerPC
- operation counters
- some reference counters
- lazy initialization
  - but for this C++ also brings `std::call_once`

If that is the case:

- wrap the code in nice encapsulations

so let us now return into the nice, cozy, sane land of the default memory model



sequentially consistent (SC)*

acquire-release

consume-release

relaxed

seq_cst

acquire
release
acq_rel

consume
release
acq_rel

relaxed

value of x? 1 or 2

Initial State:

```
  f1, f2 = false, x = 0
```

Thread #1:

```
  f1.store(true);          // A
```

Thread #2:

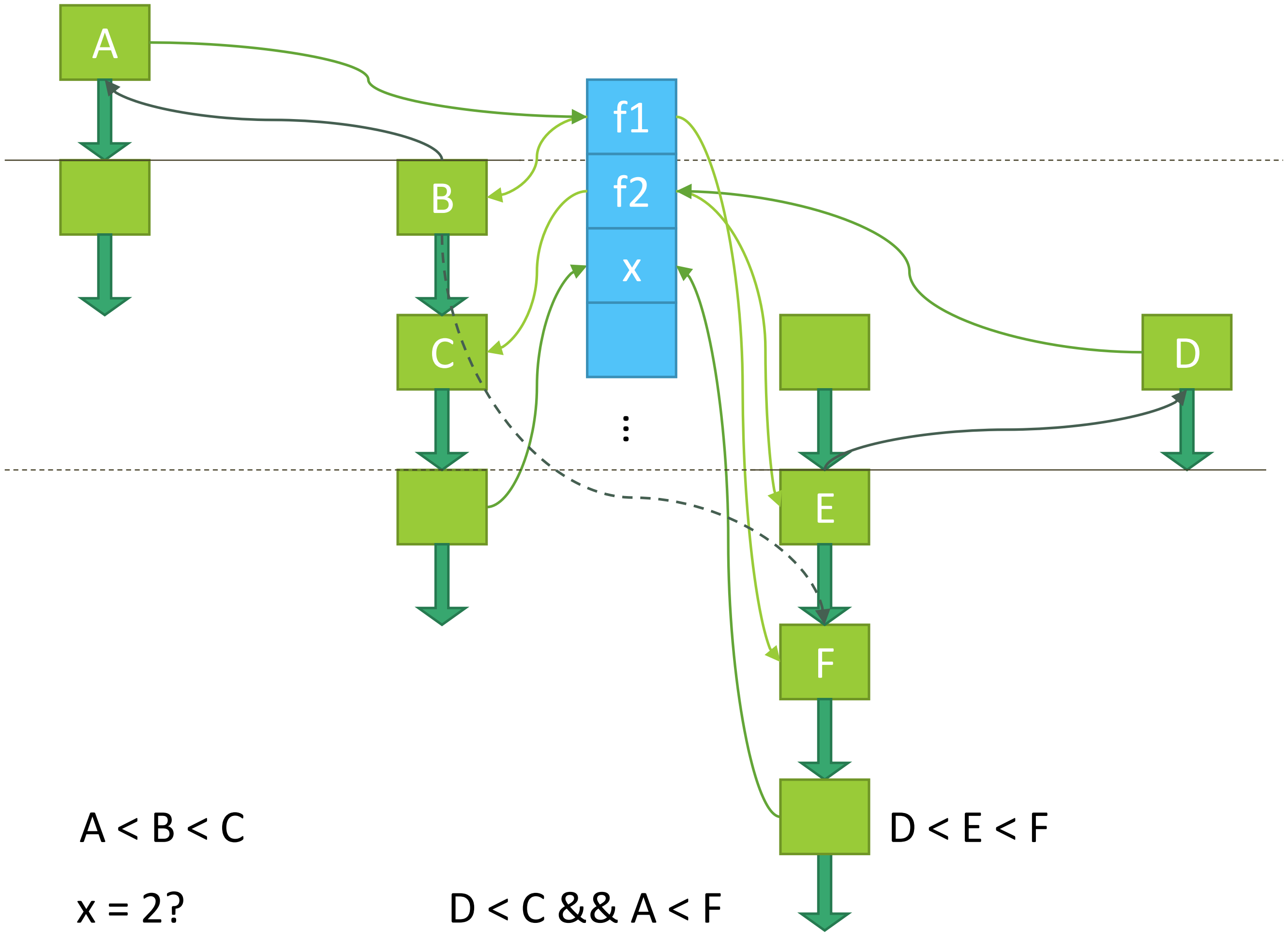
```
  while (!f1.load()) ;     // B
  if (f2.load()) { ++x;}   // C
```

Thread #3:

```
  while (!f2.load()) ;     // E
  if (f1.load()) { ++x;}   // F
```

Thread #4:

```
  f2.store(true);          // D
```



A < B < C

D < E < F

x = 2? D < C && A < F

x = 1? D > C XOR A > F

x = 0? D > C && A > F ⇒ ⚡

value of x? 1 or 2

Initial State:

```
f1, f2 = false, x = 0
```

Thread #1:

```
f1.store(true);         // A
```

Thread #2:

```
while (!f1.load()) ;    // B
if (f2.load()) { ++x;}  // C
```

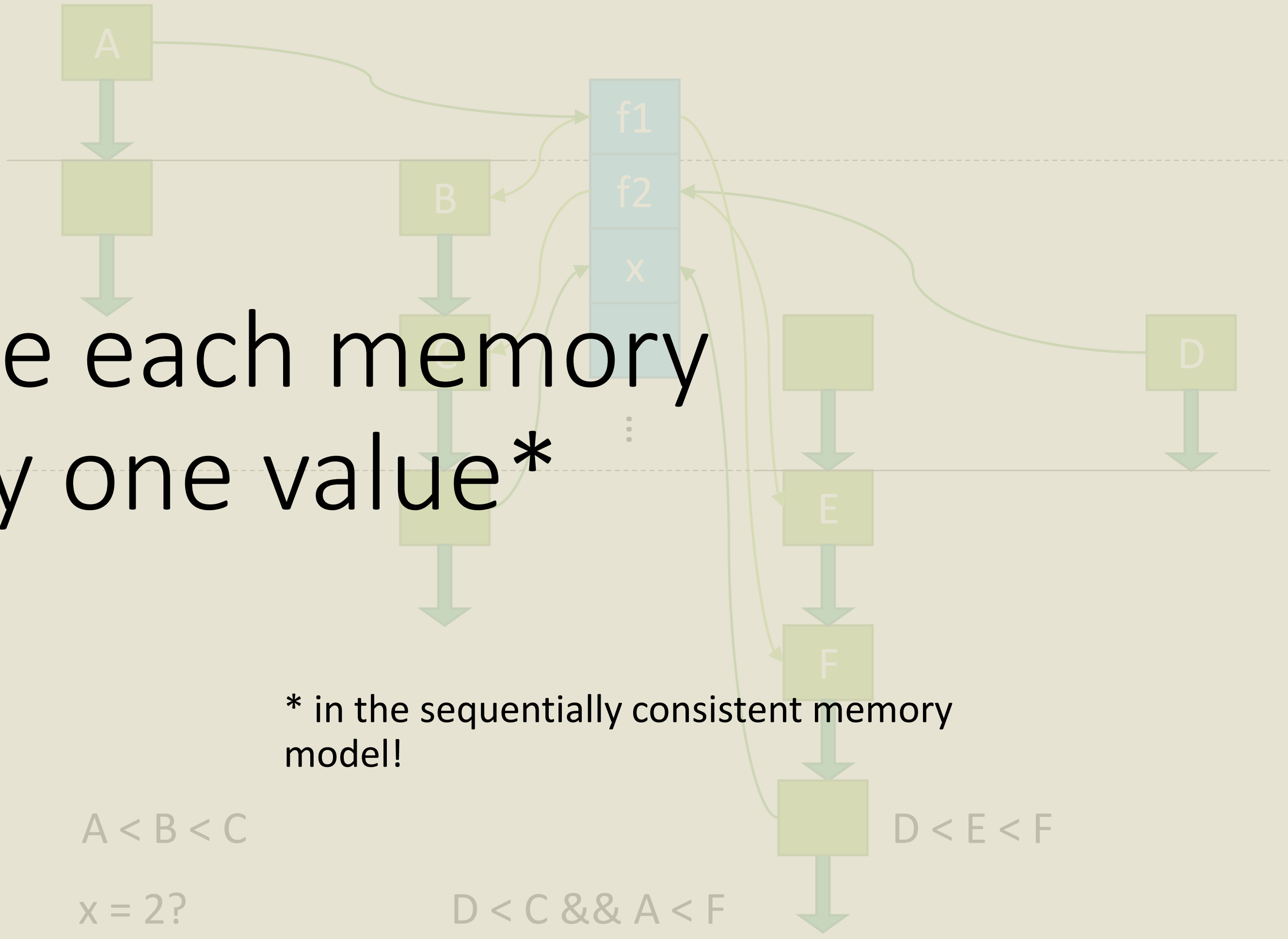
Thread #3:

```
while (!f2.load()) ;    // E
if (f1.load()) { ++x;}  // F
```

Thread #4:

```
f2.store(true);         // D
```

At any given time each memory location has only one value*



A < B < C

D < E < F

x = 2?	D < C && A < F
x = 1?	D > C XOR A > F
x = 0?	D > C && A > F ⇒ ⚡

* in the sequentially consistent memory model!

Wrap up

- The C++ memory model allows us to reason about multithreaded code
- It is gives reasonable guarantees to implement performant algorithms
- It allows us to derivate from the default model if we need to

Questions?

Bibliography

- C++ Concurrency in Action – Anthony Williams – 2012
- Atomic Weapons – Herb Sutter – 2012
- Pershing on Programming – Jeff Pershing – http://preshing.com accessed Dec. 2013
- ISO C++ Working Draft N3337 – 2012
- Foundations of the C++ Concurrency Memory Model – H. Boehm, S. V. Adve – 2008
- How to make a Multiprocessor Computer that correctly executes Multiprocess Programs – Leslie Lamport – 1979

Thank You

C++ Memory Model

Valentin Ziegler

Fabio Fracassi

think-cell

Software GmbH