C++ Memory Model

Valentin Ziegler

Fabio Fracassi

Meeting C++ Berlin, December 6th, 2014

think-cell

The machine does not execute the code you wrote…

How your code is executed

Code

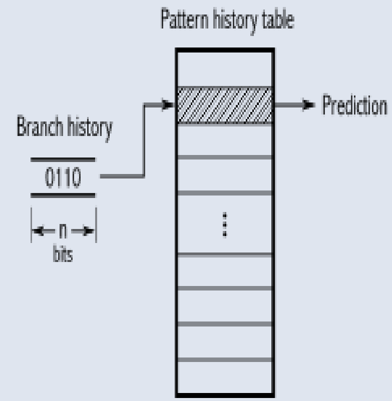
```
void foo(int n, int m)
{
   int x=1;
   for (
     int i=0;
     i<n;
     ++i
   ){
      if (m<0) {
          x -= x*m;
      }
   }
  return x;
}
```

Compiler

```
mov     esi,1
test    ecx,ecx
jle     011F1301
lea     ebx,[ebx]
mov     eax,1
sub     eax,edx
imul    esi,eax
dec     ecx
jne     011F12F0
...
```

- Register allocation
- Loop unswitching

CPU



- Branch prediction
- Out of order exec.

Cache

000 001 222 003	004 005 006 007
008 999 AAA 00B	00C 00D 00E 00F

- Prefetching
- Buffering

Memory

000
001
002
003

004
005
006
007

008
009
00A
00B

00C
00D
00E
00F

010
011
012
013

Memory

Optimization

How your code is executed

Single thread execution model (C++03):

- Program will behave ***as-if*** it was yours:
  Result is the same as if operations were executed in the order specified by the program

- We can not observe optimizations performed by the system

Two threads of execution?

```
bool f1 = false; bool f2 = false;
```

Thread #1

```
f1 = true;
if (!f2) {
  // critical section
}
…
```

Thread #2

```
f2 = true;
if (!f1) {
  // critical section
}
…
```

- Optimizations become observable
- Optimizations may break “naive” concurrent algorithms

Memory Model

- Describes the interactions of threads through memory and their shared use of data.

- Tells us if our program has well defined behavior.

- Constrains code generation for compiler

The C++ Memory model

C++ Memory Model Basics

Data Races, Sequential Consistency, Synchronization

Meddling with Memory Order

Relaxed Atomic Operations, and subtle Consequences

Data Race

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

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

conflicting action [intro.multithread(1.10)/4]

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

data race [intro.multithread(1.10)/21]

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

Data Race

memor

an ob
non-z

conflic

two (
locati

data ra

two *c*
happ

```
int  i;
char c;
int  a:5,
     b:7;
X*   p;
```

Data Race

memor

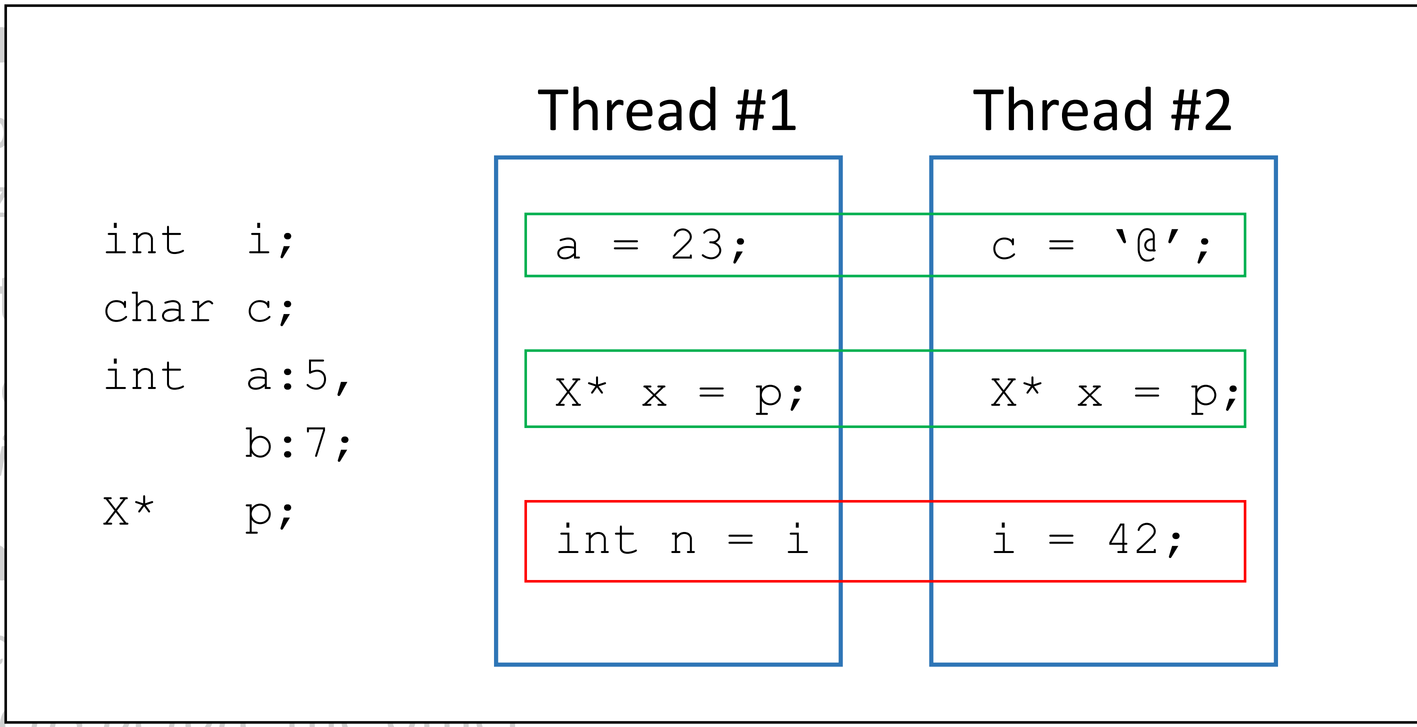
an ob
non-z

conflic

two (
locati

data ra

two c
happens before the other.



Data Race

memo

an ob
non-z

conflic

two (
locati

data ra

two *c*
happens before the other.

data race == undefined behavior !

Sequential Consistency

sequential consistency [Leslie Lamport, 1979]

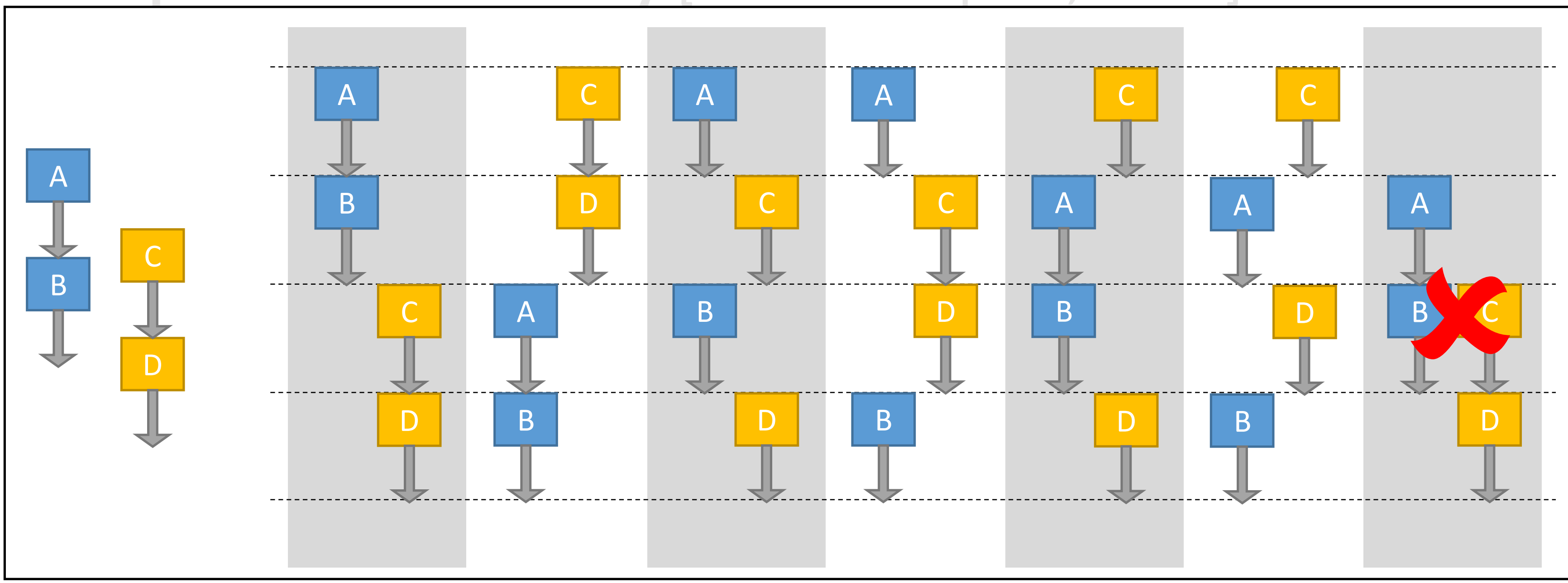
the result of any execution is the same *as-if*

1. the operations of all threads are executed in some sequential order

2. the operations of each thread appear in this sequence in the order specified by their program

Sequential Consistency

sequential consistency [Leslie Lamport, 1979]



13

The C++ memory model

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

Here is the deal:

- We take care our program does not contain data races
- The system guarantees sequentially consistent execution

synchronize (the easy way)…

Locks

Mutually exclusive execution of critical code blocks

```
std::mutex mtx;
{
        mtx.lock();
        // access shared data here
        mtx.unlock();
}
```

Mutex provides inter-thread **synchronization**:

`unlock()` synchronizes with calls to

`lock()` on the **same** mutex object.

Locks

Mutually exclusive execution of critical code blocks

```
std::mutex mtx;
{
        mtx.lock();
        // access shared data here
        mtx.unlock();
}
```

Mutex provides inter-thread **synchronization**:

`unlock()` synchronizes with calls to

`lock()` on the **same** mutex object.

Locks

Mutually exclusive execution of critical code blocks

```
std::mutex mtx;
{
        std::lock_guard<std::mutex> lg(mtx);
        // access shared data here
        // lg destructor releases mtx
}
```

Mutex provides inter-thread **synchronization**:

`unlock()` synchronizes with calls to

`lock()` on the **same** mutex object.

Synchronize using Locks

```
std::mutex mtx;    bool bDataReady=false;
```

Thread #1

```
{
  mtx.lock();
  PrepareData();
  bDataReady=true;
  mtx.unlock();
}
```

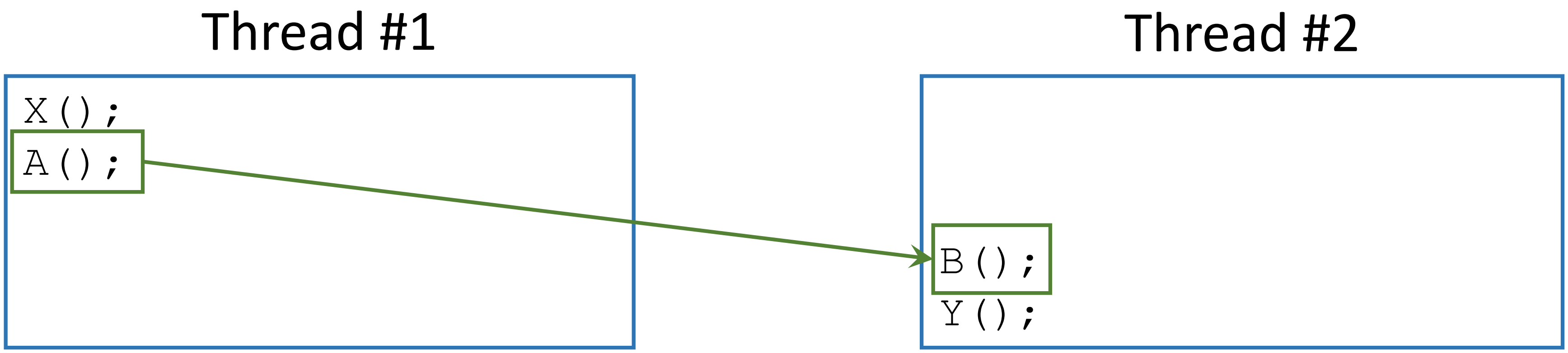
Thread #2

```
{
  mtx.lock();
  if (bDataReady) {
    ConsumeData();
  }
  mtx.unlock();
}
```

"Simplistic view" on locking: Critical code cannot run in both threads "simultaneously"

What about synchronization?

- The C++ standard identifies certain operations to be synchronizing operations.



- If `A()` synchronizes with `B()`, then `X()` happens before `Y()`.

Locks

Mutual exclusive execution of critical code blocks

```
std::mutex mtx;
{
        mtx.lock();
        // access shared data here
        mtx.unlock();
}
```

Mutex provides inter-thread **synchronization**:

`unlock()` synchronizes with calls to

`lock()` on the **same** mutex object.

Synchronize using Locks

```
std::mutex mtx;    bool bDataReady=false;
```

Thread #1

```
{
  mtx.lock();
  PrepareData();
  bDataReady=true;
  mtx.unlock();
}
```

`mtx.unlock()` **synchronizes with** `mtx.lock()`
`PrepareData()` **happens before** `ConsumeData()`

Thread #2

```
{
  mtx.lock();
  if (bDataReady) {
    ConsumeData();
  }
  mtx.unlock();
}
```

Synchronize using Locks

```
std::mutex mtx;    bool bDataReady=false;
```

Thread #1

```
PrepareData(); // once
{
  mtx.lock();
  bDataReady=true;
  mtx.unlock();
}
```

Thread #2

```
bool b;
{
  mtx.lock();
  b=bDataReady;
  mtx.unlock();
}
if (b) ConsumeData();
```

Proper synchronization,
if `PrepareData()` is <u>never</u> executed again.

Clever ?

```
std::mutex mtx;    bool bDataReady=false;
```

Thread #1

```
PrepareData(); // once
{
  mtx.lock();
  bDataReady=true;
  mtx.unlock();
}
```

Thread #2

```
if (!mtx.try_lock()){
  // l33t optimization:
  // thread 1 should
  // be done with
  // PrepareData :-)
  ConsumeData();
}
```

No synchronization.

Data Race!

`std::atomic<>`

- “Data race free” variable, e.g., `std::atomic<int>`
- (by default) provides inter-thread **synchronization**:

  a `store` synchronizes with operations that `load` the stored value.

- (by default) sequential consistency
- Needs hardware support
  (not all platforms provide lock-free atomics)

std::atomic<>

In C++, this is spelled `std::atomic`, not `volatile`!

mic<int>

onization:

a `store` synchronizes with operations that `load` the stored value.

- (by default) sequential consistency
- Needs hardware support
  (not all platforms provide lock-free atomics)

Synchronize using atomics

```
std::mutex mtx;    std::atomic<bool> bDataReady(false);
```

Thread #1

```
PrepareData(); // once
bDataReady.store(true);
```

Thread #2

```
if(bDataReady.load()){
  ConsumeData()
}
```

Proper synchronization,
if `PrepareData()` is <u>never</u> executed again.

Excursion: lock-free programming

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

Target (head) — **Expected** (newNode->next) — **Desired** (newNode)

Are we there yet?

Avoid data races and you will be fine

- Synchronize correctly
- Implement lock-free data structures as described in your favorite computer science text book

The C++ memory model guarantees sequential consistency

- as does the memory model of Java and C#

The C++ Memory model

C++ Memory Model Basics

Data Races, Sequential Consistency, Synchronization

Meddling with Memory Order

Relaxed Atomic Operations, and subtle Consequences

`std::atomic<>`

- (by default) provides inter-thread **synchronization**:

  a `store` synchronizes with operations that `load` the stored value.

- (by default) sequential consistency

Memory Order

Why only have one memory model when we can have a mix of 3 (and a half)?

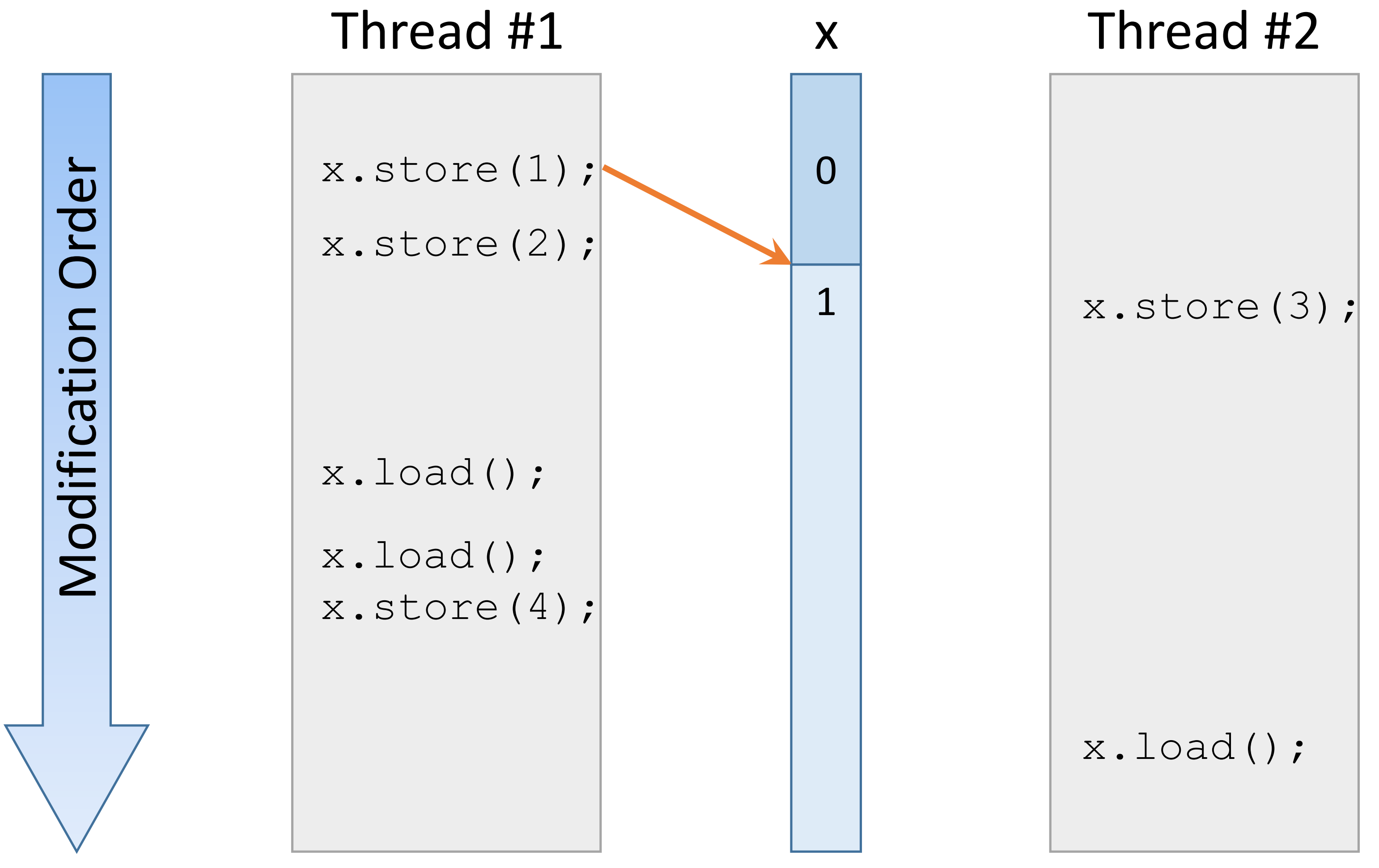
Relaxed memory ordering

What is relaxed memory order

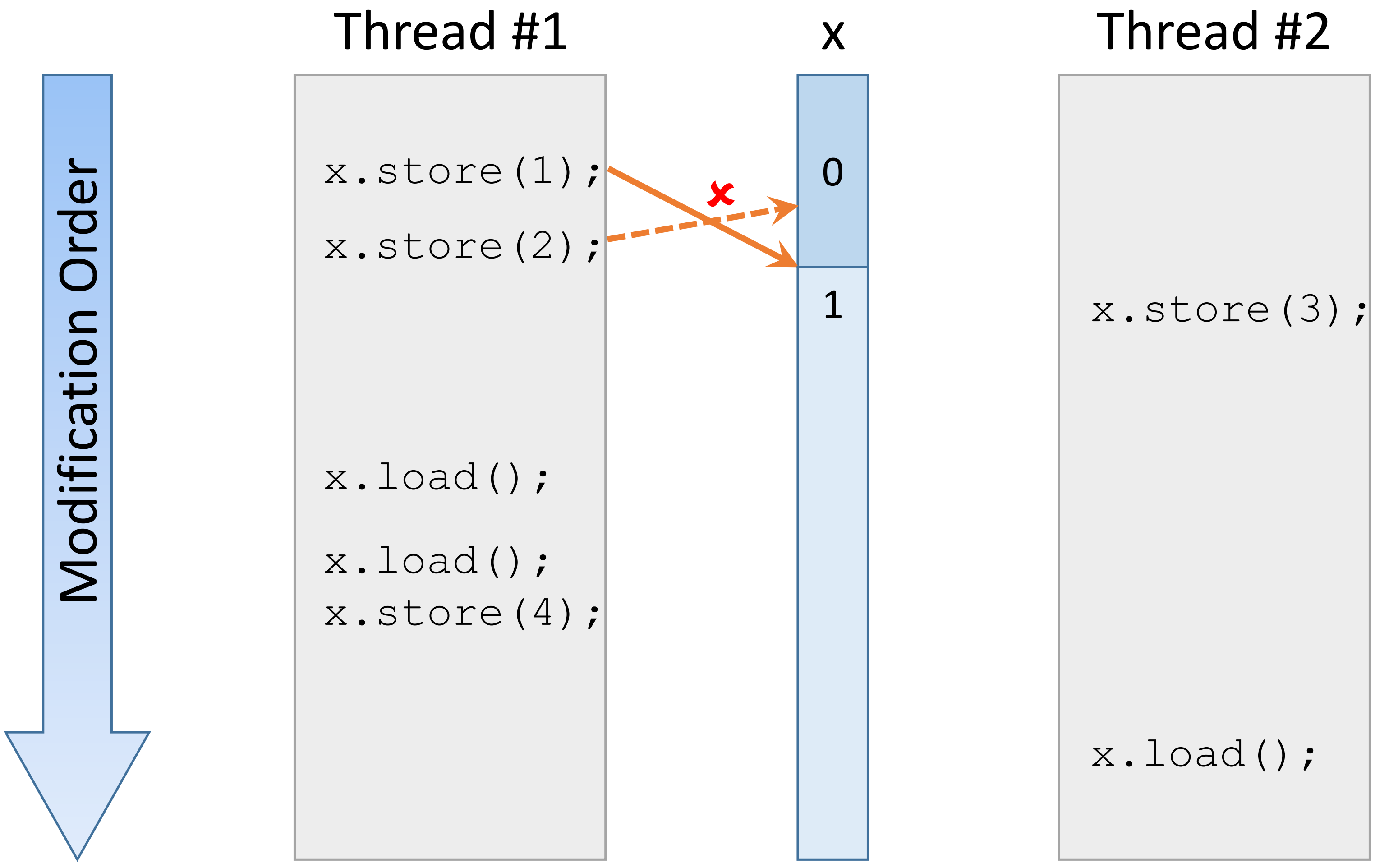
```
std::atomic<int> x;

x.store(42, memory_order_relaxed);
x.load(memory_order_relaxed);

x.compare_exchange_weak(
  n, 42, memory_order_relaxed
);
```

- Each memory location has a total **modification order** (however, this order cannot be observed directly)
- Memory operations performed by the **same thread** on the **same memory location** are not reordered with respect to the modification order.

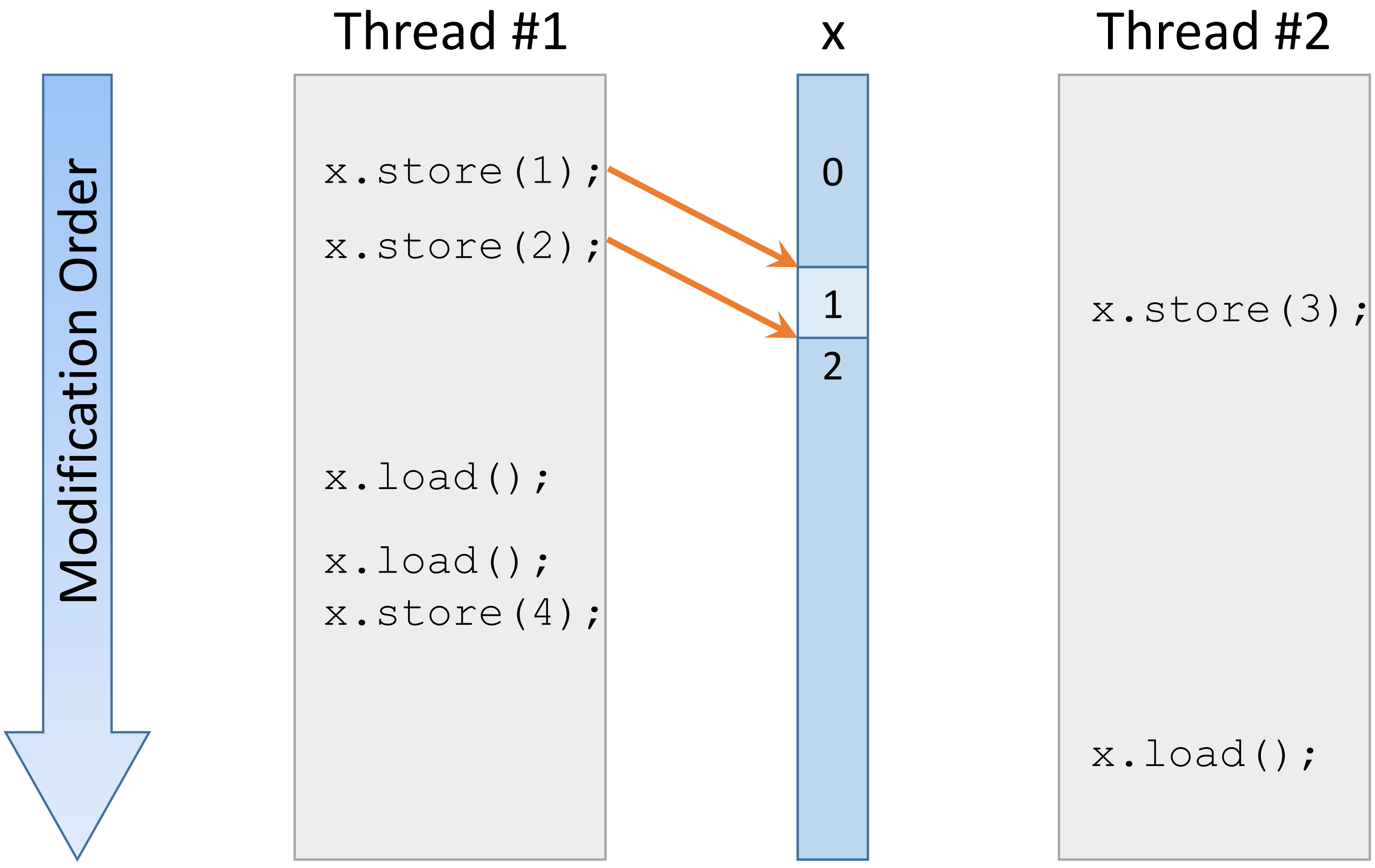
Relaxed load and store



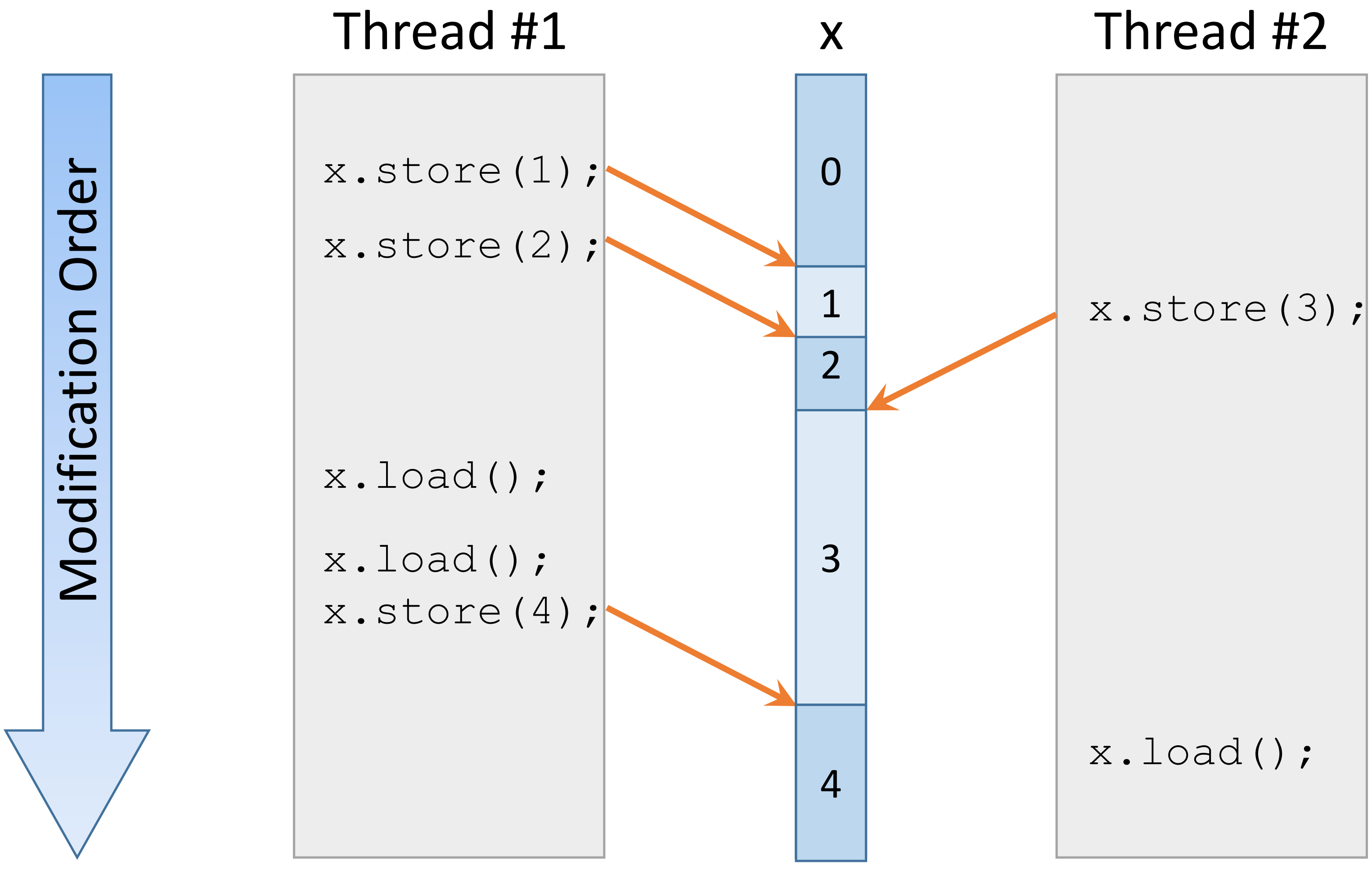
Relaxed load and store



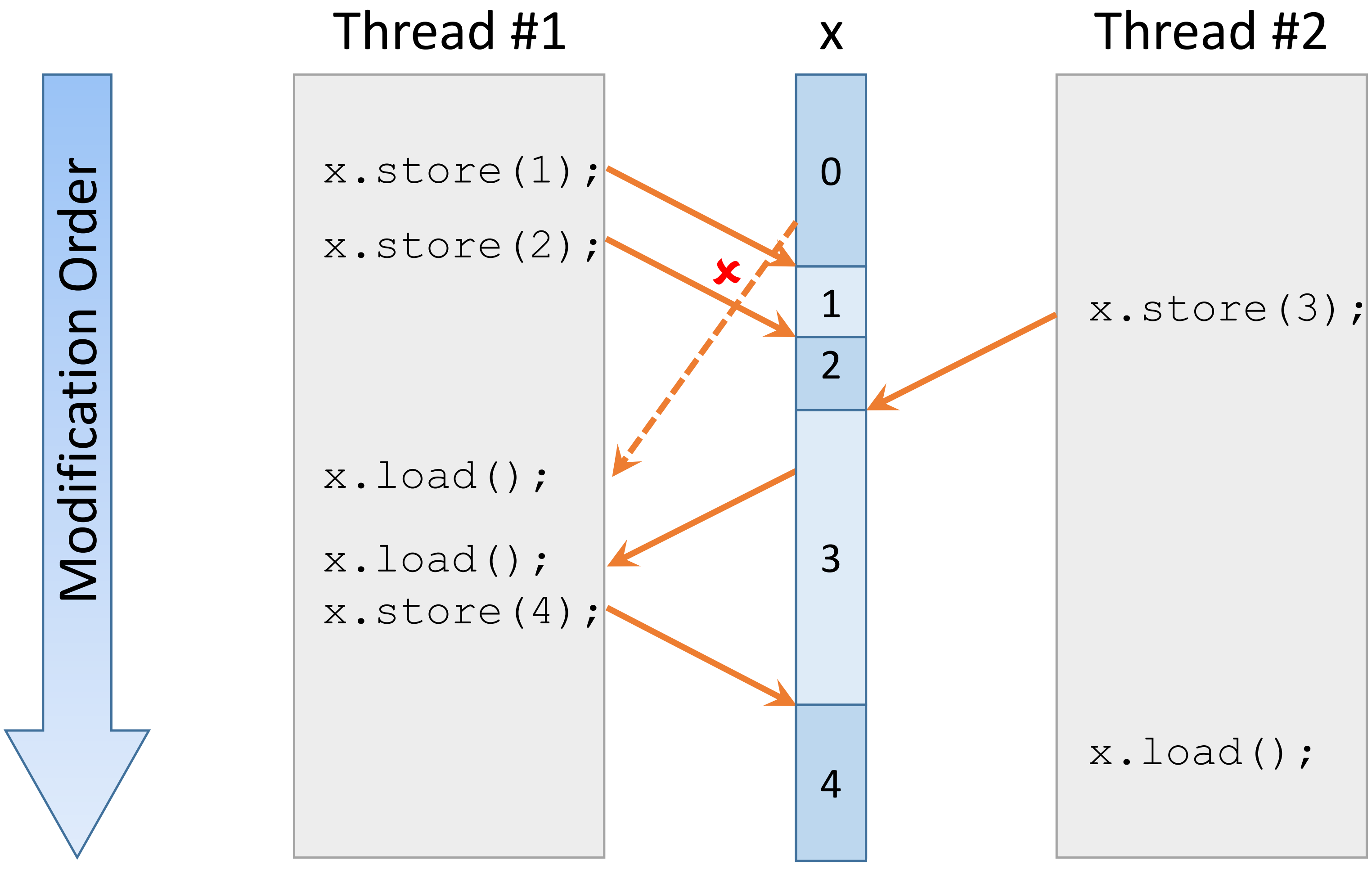
Relaxed load and store



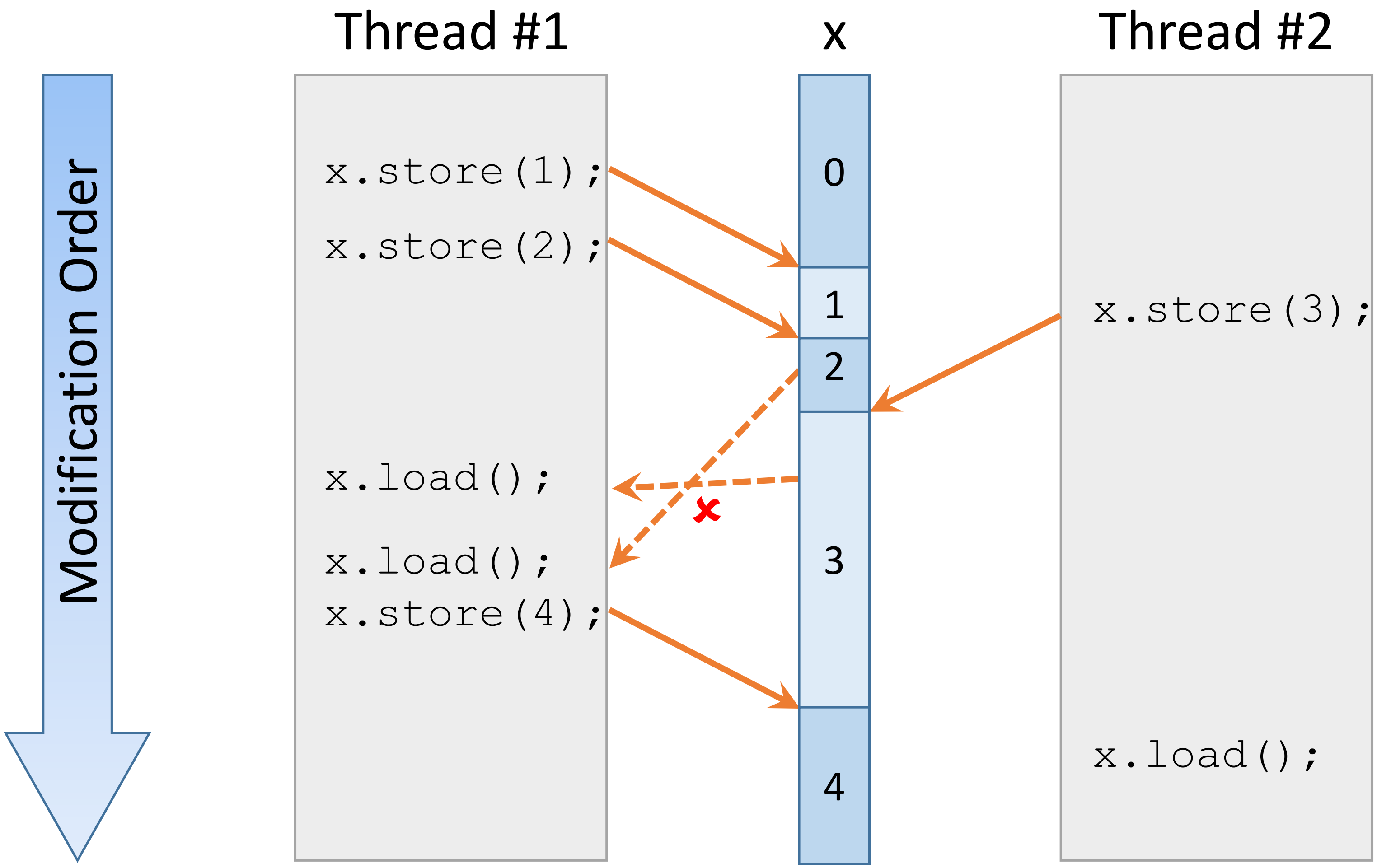
Relaxed load and store



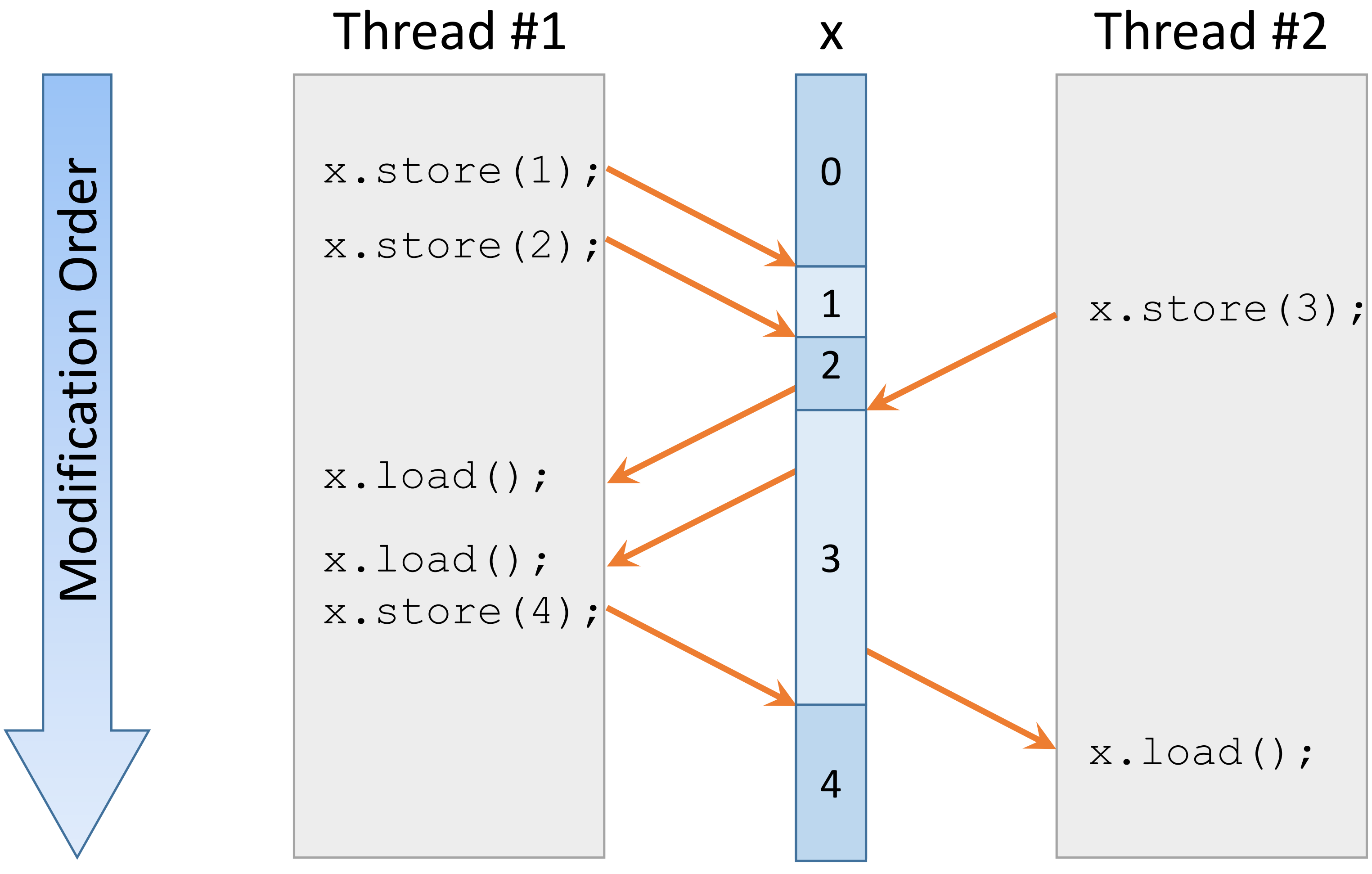
Relaxed load and store



Relaxed load and store



Relaxed load and store



Relaxed load and store

Safe ?

```
atomic<bool> f=false;
atomic<bool> g=false;
```

Thread #1:

```
f.store(true, memory_order_relaxed);
g.store(true, memory_order_relaxed);
```

Thread #2:

```
while(!g.load(memory_order_relaxed));
assert(f.load(memory_order_relaxed));
```

Relaxed load and store

Safe ?

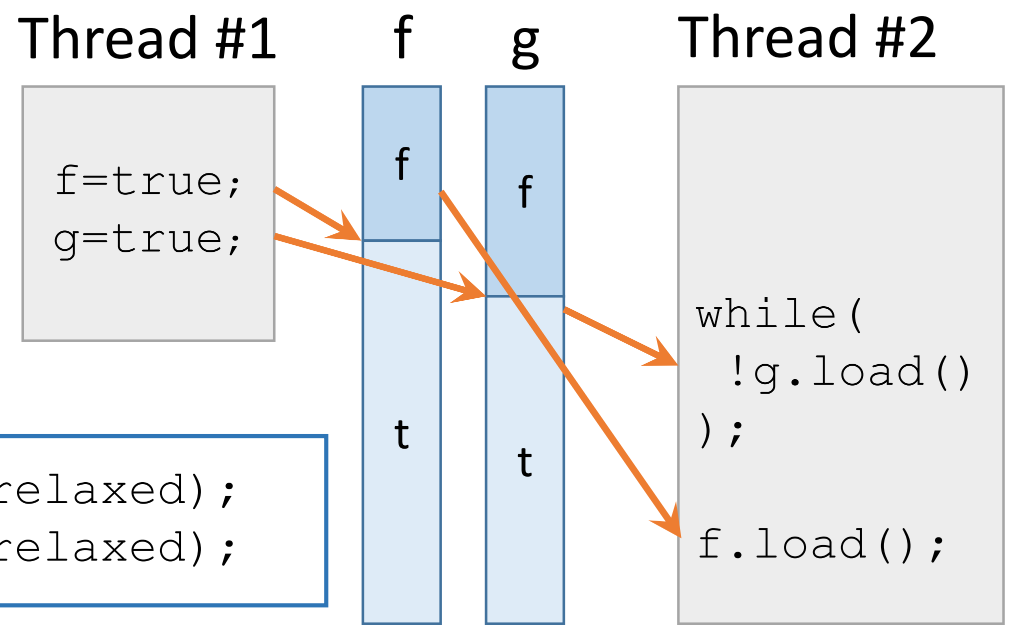
```
atomic<bool> f=false;
atomic<bool> g=false;
```

Thread #1:

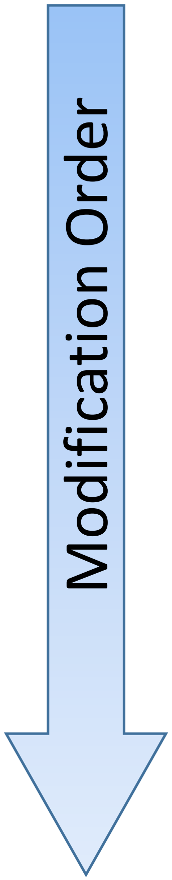
```
f.store(true, memory_order_relaxed);
g.store(true, memory_order_relaxed);
```

Thread #2:

```
while(!g.load(memory_order_relaxed));
assert(f.load(memory_order_relaxed));
```



Relaxed read-modify-write



lock-free `++x`

```
int oldVal=x.load();
```

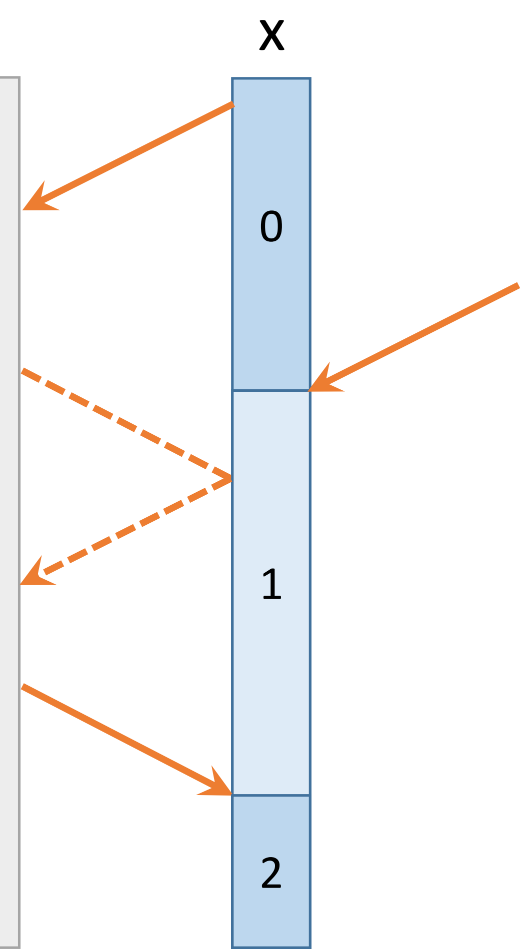
Sets oldVal to 0

```
x.compare_exchange_weak(
   oldVal,     // expected
   oldVal+1,   // desired
   memory_order_relaxed);
```

Returns false

Sets oldVal to 1

```
x.compare_exchange_weak(
   oldVal,     // expected
   oldVal+1,   // desired
   memory order relaxed);
```



Safe? Yes. Progress ?

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

Worker thread #1,#2, …:

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

Built-in for

```
int oldVal=c.load();
while(!c.compare_exchange_weak(
   oldVal,     // expected
   oldVal+1    // desired
   memory_order_relaxed
);
```

Main thread:

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

Safe? Yes. Progress ? Yes.

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

Worker thread #1,#2, …:

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

Observing thread ("progress bar"):

```
for (int old_c = 0;;) {
  int c_now = c.load(memory_order_relaxed);
  assert(old_c <= c_now); ✓
  old_c = c_now;
}
```

Main thread:

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

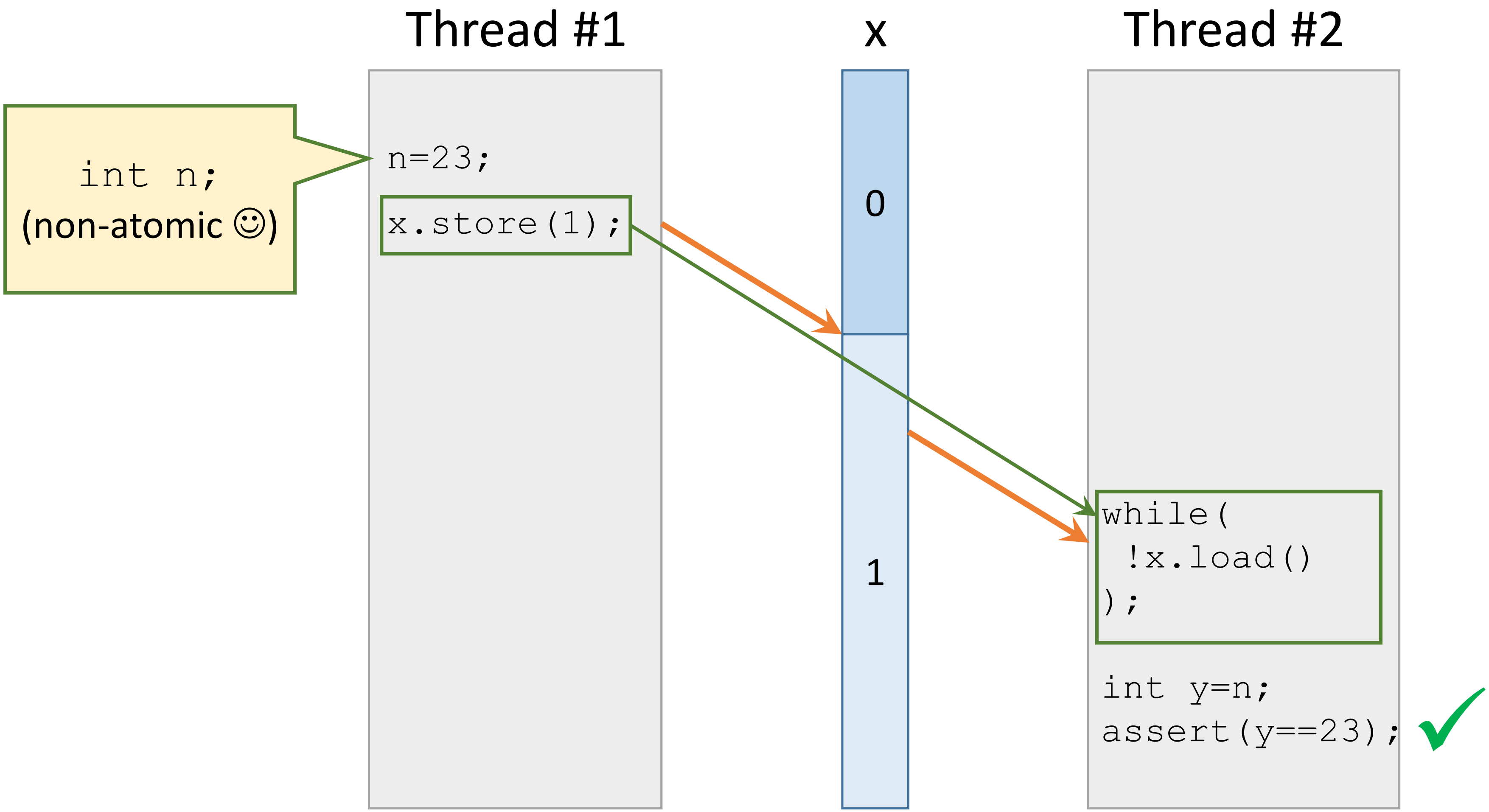
The acquire/release model

What does acquire/release mean

```
x.store(42, memory_order_release);
x.load(memory_order_acquire);
```

- a **store-release** operation **synchronizes** with all **load-acquire** operations reading the stored value.
- All Operations in the releasing thread preceding the store-release **happen-before** all operations following the load-acquire in the acquiring thread.

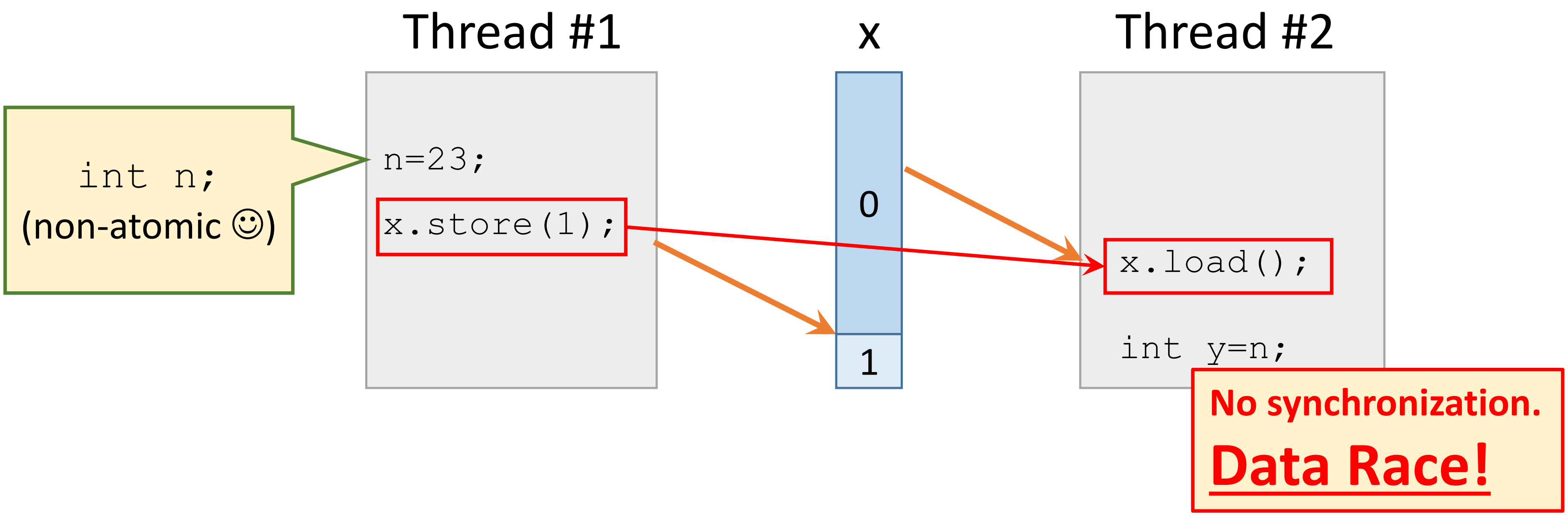
Store-relase and load-acquire



A note of caution!

"Synchronizes with" relation:

- Refers to operations at **runtime**.
- **NOT** about statements in the source code!



Is the answer 42 ? Yes.

```
atomic<bool> f=false;
atomic<bool> g=false;
int n;
```

Thread #1:

```
n = 42;
f.store(true, memory_order_release);
```

Thread #2:

```
while(!f.load(memory_order_acquire));
g.store(true, memory_order_release);
```

Thread #3:

```
while(!g.load(memory_order_acquire));
assert(42 == n);
```

The consume/release model

What does consume mean

```
x.store(42, memory_order_release);
x.load(memory_order_consume);
```

- “Light version” of acquire/release
- All Operations in the releasing thread preceding the store-release **happen-before** an operation X in the consuming thread if X depends on the value loaded.

Who has the answer? `x->i`

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

Thread #1:

```
n = 42;
auto x = new X;
x->i = 42;
px.store(x, memory_order_release);
```

Thread #2:

```
X* x;
while(!x=px.load(memory_order_consume));
assert(42 == x->i); ✓
assert(42 == n); ✗ Data Race !
```

Typical examples for “*X depends on the value loaded*”

- X dereferences a pointer that has been loaded
- X is accessing array at index which has been loaded

Acquire/release provides very strong guarantees.

Do we still need more?

Who asked for sequential consistency ?

Dekker’s algorithm revisited

```
atomic<bool> f1=false;
atomic<bool> f2=false;
```

Thread #1:

```
f1.store(true, memory_order_release);
if (!f2.load(memory_order_acquire)) {
   // critical section
}
```

Thread #2:

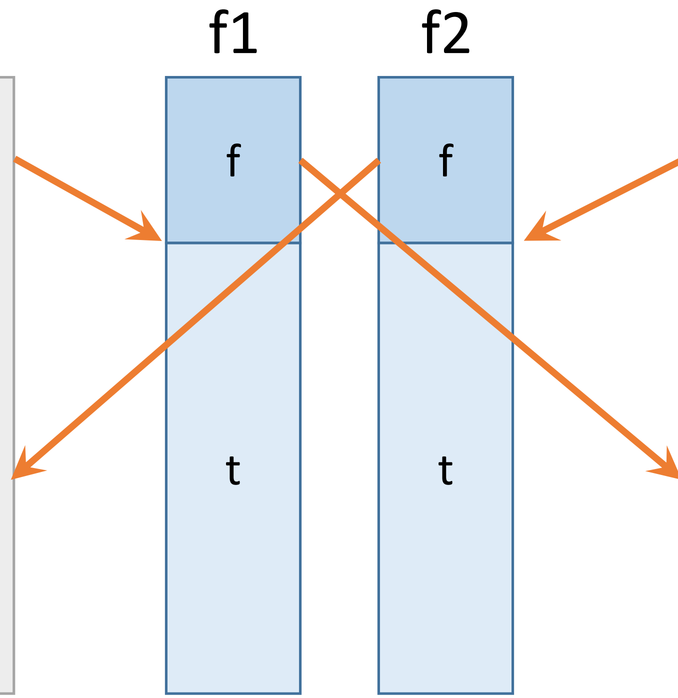
```
f2.store(true, memory_order_release);
if (!f1.load(memory_order_acquire)) {
   // critical section
}
```

?

Dekker’s algorithm revisited

Thread #1

```
f1=true;



if (!f2.load()) {
  // critical
  // section
}
```



Thread #2

```
f2=true;



if (!f1.load()) {
  // critical
  // section
}
```

Oh noes!

Back to ~~sanity~~ sequential consistency

sequentially consistent (SC)*

acquire-release

consume-release

relaxed

`seq_cst`

`acquire`
`release`
`acq_rel`

`consume`
`release`
`acq_rel`

`relaxed`

Dekker's algorithm done right.

```
atomic<bool> f1=false;
atomic<bool> f2=false;
```

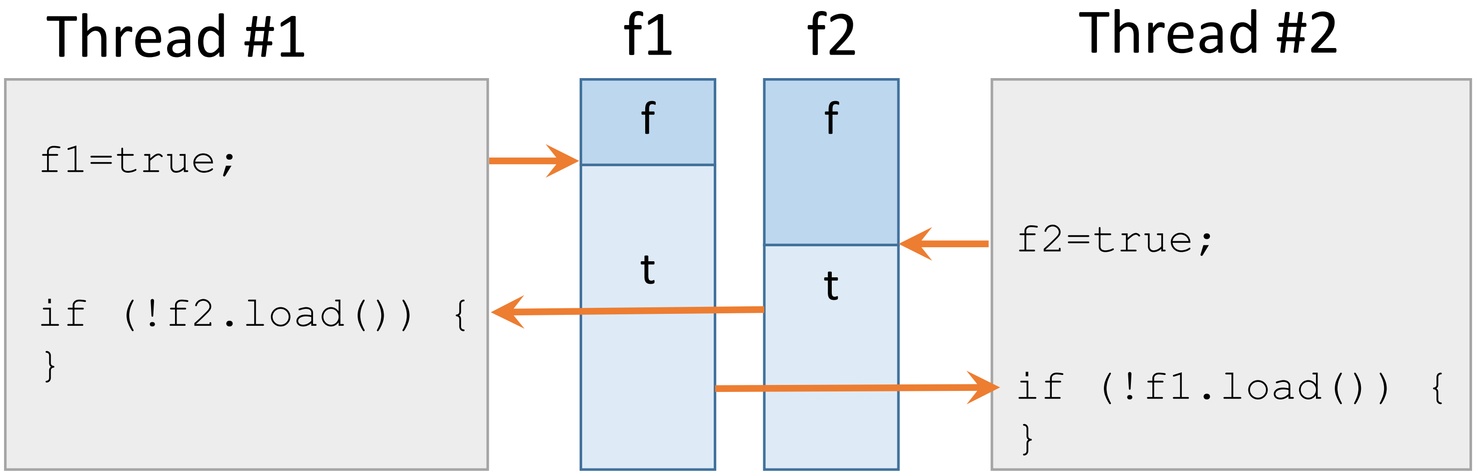
Thread #1:

```
f1.store(true, memory_order_seq_cst);
if (!f2.load(memory_order_seq_cst)) {
   // critical section
}
```

Thread #2:

```
f2.store(true, memory_order_seq_cst);
if (!f1.load(memory_order_seq_cst)) {
   // critical section
}
```

Dekker's algorithm done right.



- Global, total order of `load` and `store` operations
- At any given time, each memory location has only one value*

* assuming there are no data races

Use-cases for non-SC atomics

- target platform is ARM (<v8) or PowerPC

- operation counters
- some reference counters
  - but then you may use `std::shared_ptr`
- lazy initialization
  - but for this C++ also brings `std::call_once`

PROFILE FIRST before meddling with memory_order!

Wrap up

- Do not write Data Races!
- The C++ Memory Model gives reasonable guarantees to implement correct, yet performant algorithms.
- It allows us to deviate from sequential consistency if we need to.

hr@think-cell.com

think-cell
Chausseestraße 8/E
10115 Berlin
Germany

Tel +49-30-666473-10
Fax +49-30-666473-19

www.think-cell.com

Bibliography

- C++ Concurrency in Action – Anthony Williams – 2012
- Atomic Weapons – Herb Sutter – 2012
- Preshing on Programming – Jeff Preshing – 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

std::atomic<> on x86/x64

	load	store	compare_exchange
memory_order_seq_cst	MOV prevent compiler optimizations	(LOCK) XCHG prevent compiler optimizations	LOCK CMPXCHG prevent compiler optimizations
memory_order_acquire **memory_order_release** **memory_order_acq_rel**	MOV prevent compiler optimizations	MOV prevent compiler optimizations	LOCK CMPXCHG prevent compiler optimizations
memory_order_relaxed	MOV	MOV	LOCK CMPXCHG

[http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html]

std::atomic<> on ARMv7

	load	store	compare_exchange
memory_order_seq_cst	ldr ; dmb prevent compiler optimizations	dmb ; str ; dmb ; prevent compiler optimizations	dmb ; LOOP ; isb prevent compiler optimizations
memory_order_acquire **memory_order_release** **memory_order_acq_rel**	ldr ; dmb prevent compiler optimizations	dmb ; str prevent compiler optimizations	(dmb;) LOOP (;isb) prevent compiler optimizations
memory_order_relaxed	ldr	str	LOOP

Subroutine LOOP :=

```
_loop:
    ldrex roldval, [rptr];
    mov rres, 0; teq roldval, rold;
    strexeq rres, rnewval, [rptr];
    teq rres, 0; bne _loop
```

[http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html]