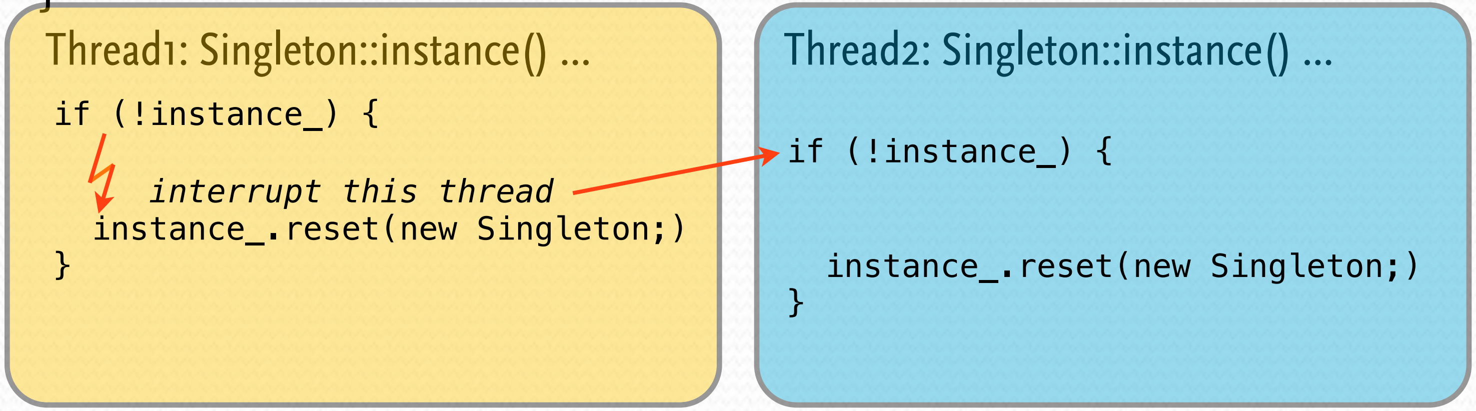
threads and singletons

In C++98 (C++03) war es NICHT möglich, Singletons thread-safe, effizient und zugleich portabel zu implementieren :-(

```
// 1. Versuch: the race begins
class Singleton {
    static std::shared_ptr<Singleton> instance_; // = 0 somewhere
    Singleton() {} // private !
public:
    static std::shared_ptr<Singleton> instance() {
        if (!instance_) { // NOT CREATED YET
            instance_.reset(new Singleton);
        }
        return instance_;
    }
};
```



Thread1: Singleton::instance() ...

```
if (!instance_) {
    interrupt this thread
  instance_.reset(new Singleton;)
}
```

Thread2: Singleton::instance() ...

```
if (!instance_) {

  instance_.reset(new Singleton;)
}
```

threads and singletons

In C++98 (C++03) war es NICHT möglich, Singletons thread-safe, effizient und zugleich portabel zu implementieren :-(

```
// 2. Versuch: lock it all
class Singleton {
    static std::shared_ptr<Singleton> instance_; // = 0 somewhere
    Singleton() {} // private !
    static std::mutex s_mutex;
public:
    static std::shared_ptr<Singleton> instance() {
        std::lock_guard<mutex> lock (s_mutex);
        if (!instance_) { // NOT CREATED YET

            instance_.reset(new Singleton);
        }
        return instance_;
    }
};
```

viel zu teuer: jeder Zugriff mit lock, nur ein einziger (der allererste) wäre nötig !

threads and singletons

In C++98 (C++03) war es NICHT möglich, Singletons thread-safe, effizient und zugleich portabel zu implementieren :-(

```
// 3. Versuch: the double checked locking - antipattern !
class Singleton {
    static std::shared_ptr<Singleton> instance_; // = 0 somewhere
    Singleton() {} // private !
    static std::mutex s_mutex;
public:
    static std::shared_ptr<Singleton> instance() {
        if (!instance_) { // NOT CREATED YET
            std::lock_guard<mutex> lock (s_mutex); // only here
            if (!instance_) { // again, with lock
                instance_.reset(new Singleton);
            }
        }
        return instance_;
    }
};
```

google for:
C++ and The Perils of Double-Checked Locking

By Scott Meyers and Andrei Alexandrescu , July 01, 2004

funktioniert leider auch nicht garantiert!

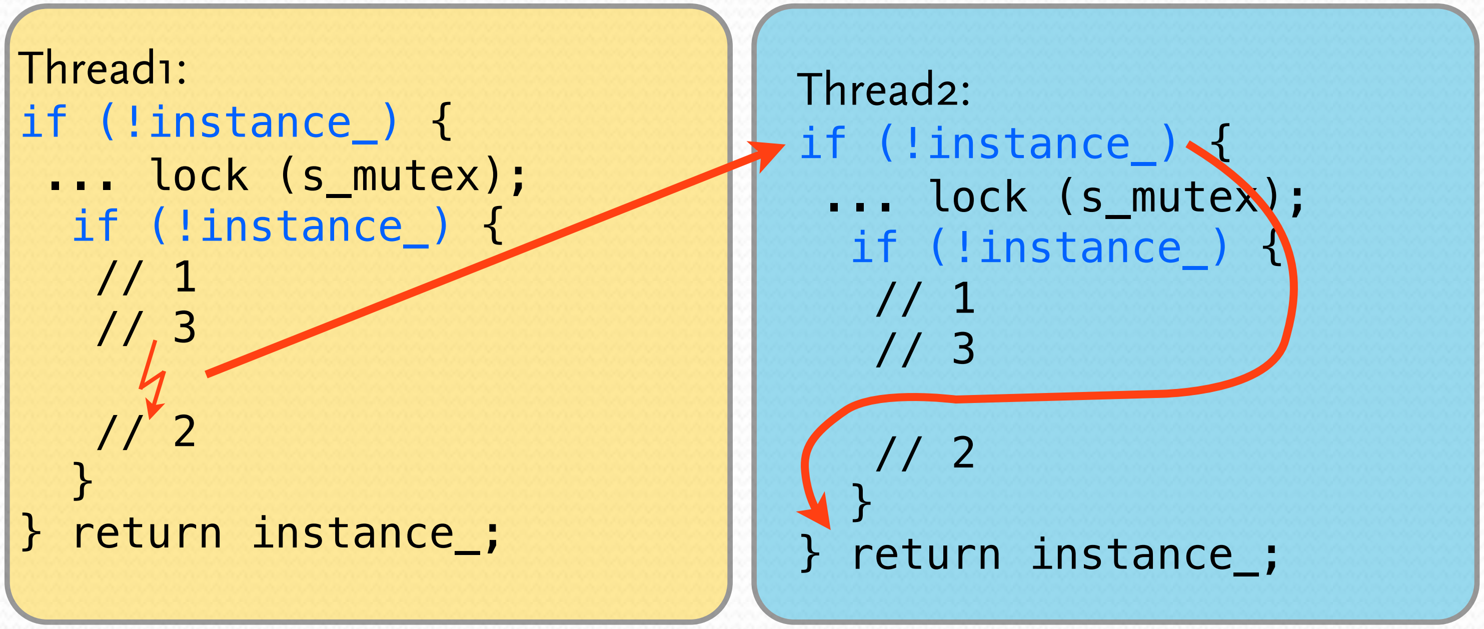
threads and singletons

3

`instance_.reset(new Singleton);` ist NICHT atomar, sondern besteht aus:

1. *allocate memory of sizeof(Singleton)*
2. *call S() in this area*
3. *assign address to instance_*

Compiler dürfen 2. und 3. aus Effizienzgründen in beliebiger Reihenfolge ausführen



Thread1:
```
if (!instance_) {
 ... lock (s_mutex);
  if (!instance_) {
   // 1
   // 3

   // 2
  }
} return instance_;
```

Thread2:
```
if (!instance_) {
 ... lock (s_mutex);
  if (!instance_) {
   // 1
   // 3

   // 2
  }
} return instance_;
```

??? not ready !!!

threads and singletons

C++11: `once_flag` und `call_once()`

```
// 4. Versuch: once and for all

#include <mutex>

class Singleton {
    static std::shared_ptr<Singleton> instance_;
    static std::once_flag oflag;
    Singleton(Args args) {} // private !
    static void safe_create() {
            instance_.reset(new Singleton(args));
    }
public:
    static std::shared_ptr<Singleton> instance() {
        std::call_once(oflag, safe_create, args); // variadic args
        return instance_;
    }
};
```

threads and singletons

C++11: static local data are thread safe

```
// even simpler –
// a Singleton without lazy creation:
class Singleton {
    Singleton(Args args) {} // private !
public:
    static Singleton& instance() {
      static Singleton instance_(args);
      return instance_;
    }
};
```

threads condition variables

Message passing between threads:
`wait` und `notify`

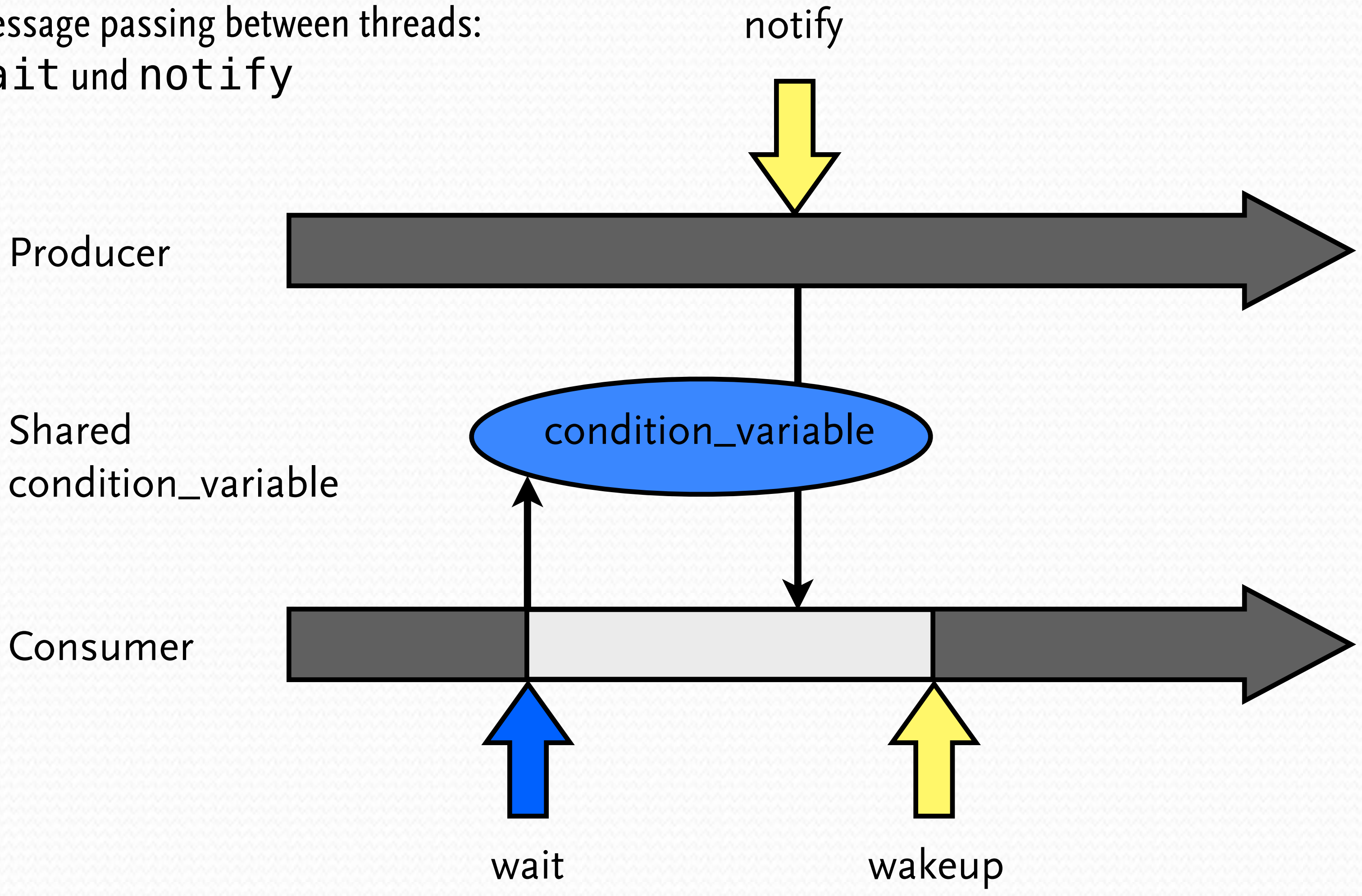
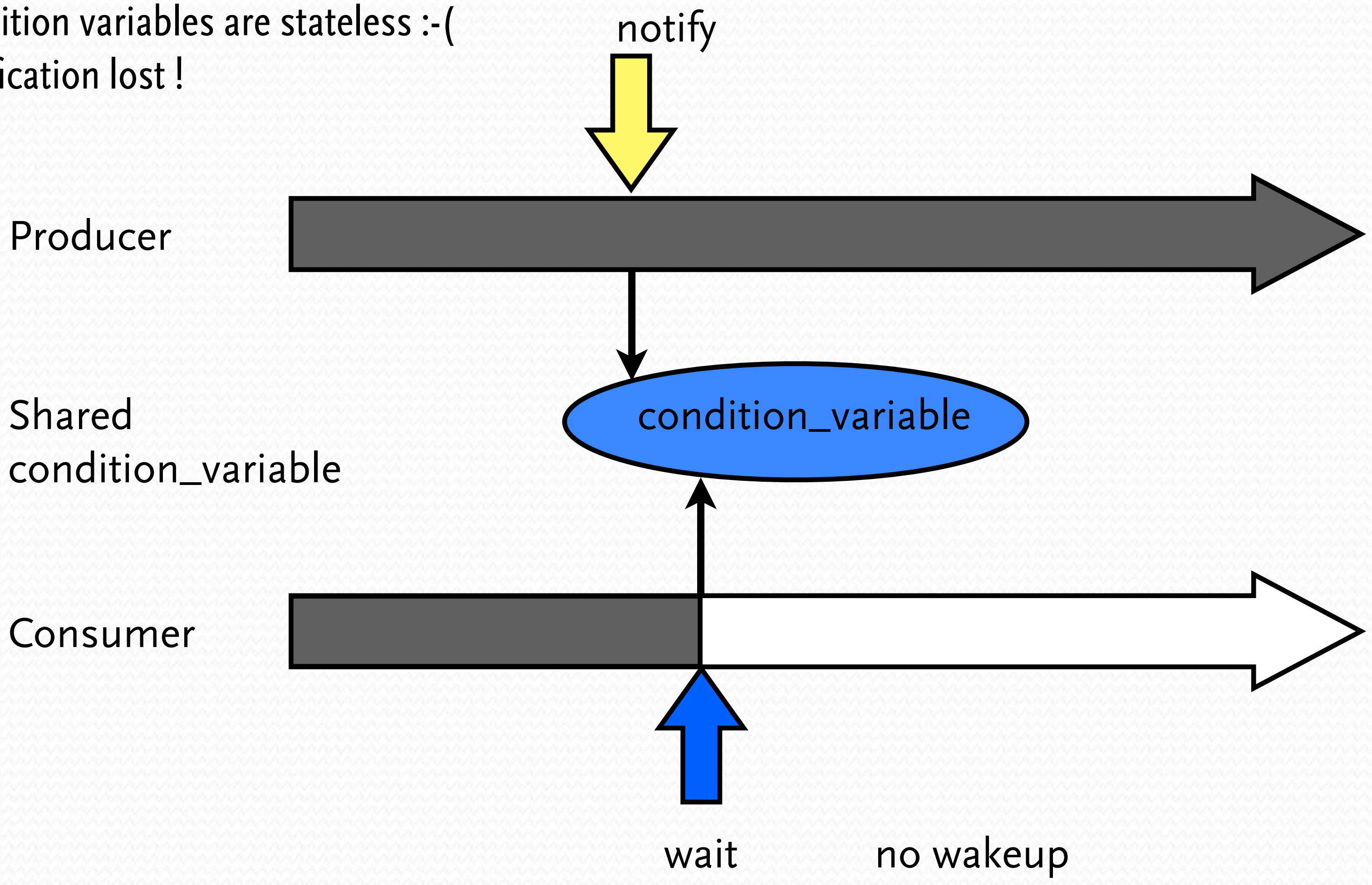


Diagramme nach Bartosz Milewski

threads condition variables

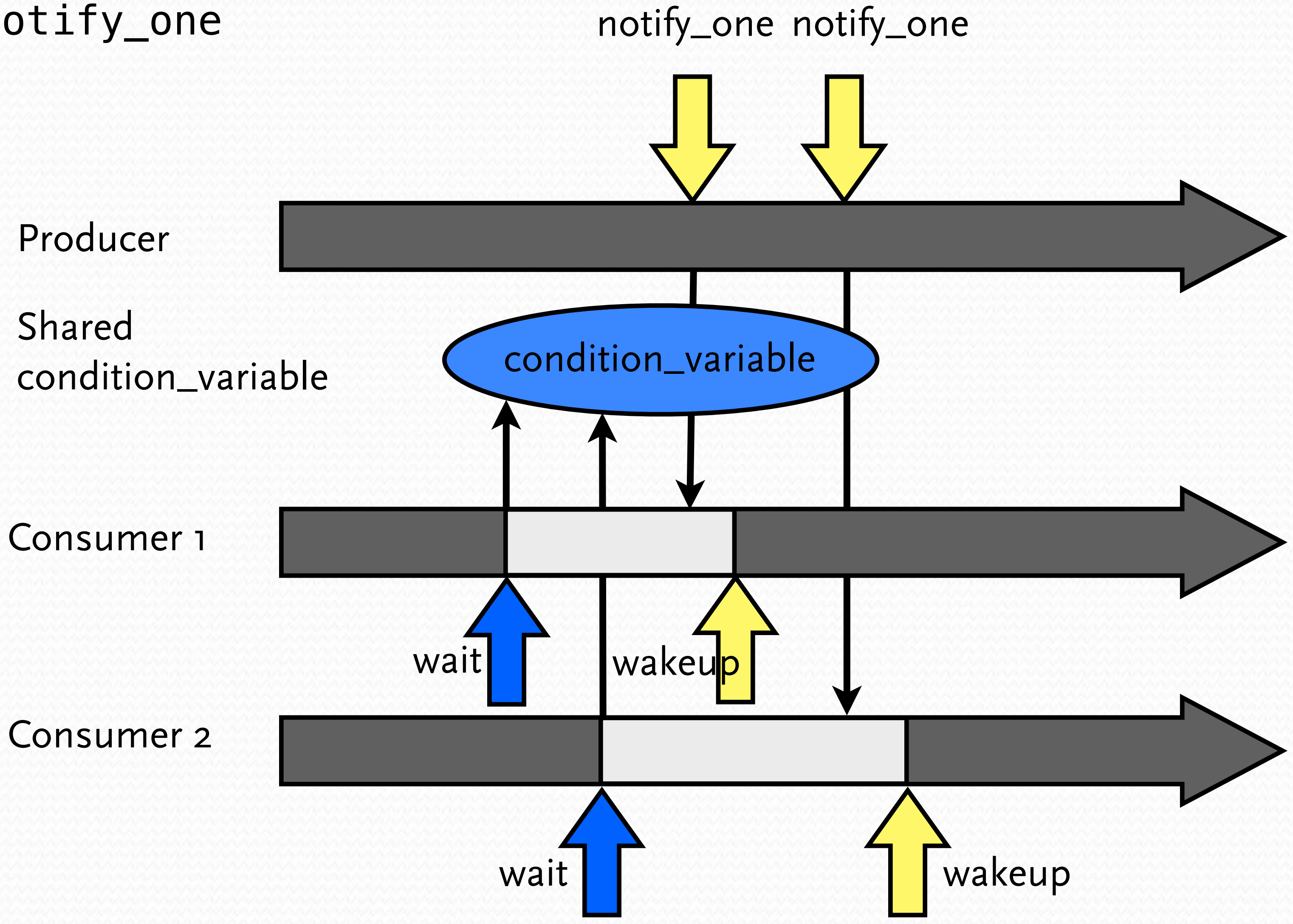
condition variables are stateless :-(

notification lost !



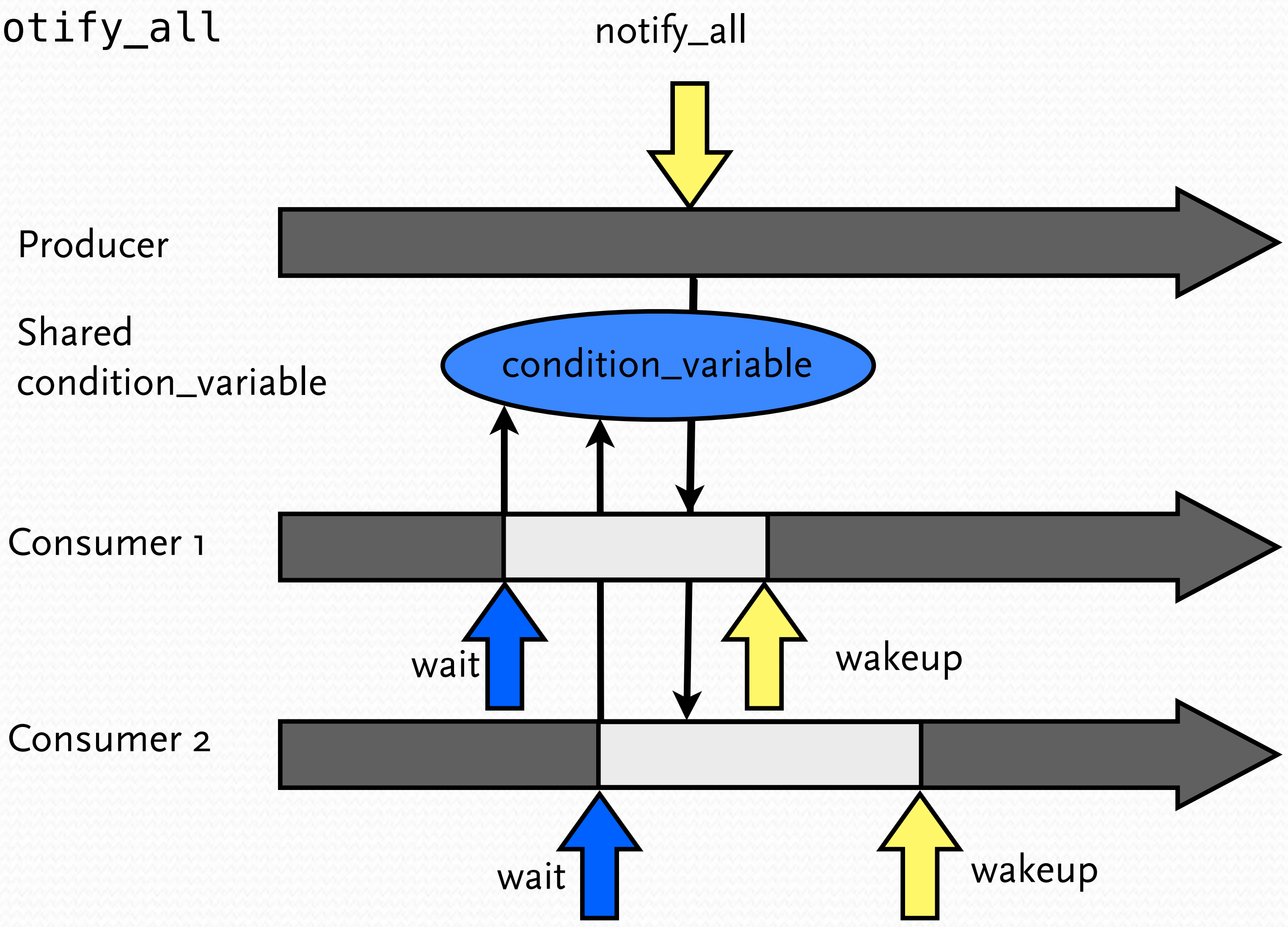
threads condition variables

`notify_one`



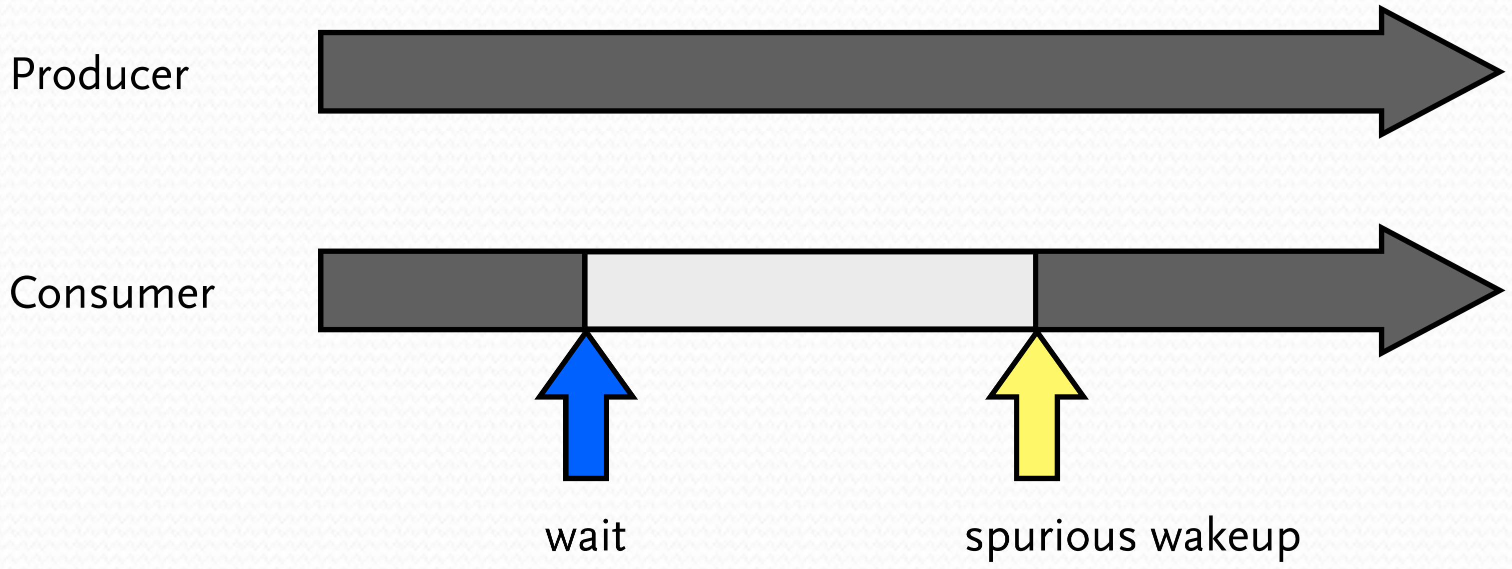
threads condition variables

notify_all



threads condition variables

spurious wakeups: Spurious wakeups may sound strange, but on some multiprocessor systems, making condition wakeup completely predictable might substantially slow all condition variable operations.



condition variables sind also recht beschränkt:
überträgt „ein Bit“ ohne Gedächtnis und nicht mal verlässlich

Wie kann man sie dennoch sinnvoll benutzen?

threads condition variables

Shared Variable Protokoll

Producer Thread

```
v = true;

cond.notify_one();
```

Consumer Thread

```
while (!v)

   cond.wait(?);
```

Locking Protokoll

```
{
 lock_guard lck(mtx);
 v = true;
}
cond.notify_one();
```

```
unique_lock lck(mtx);
while (!v)
   cond.wait(lck);
```

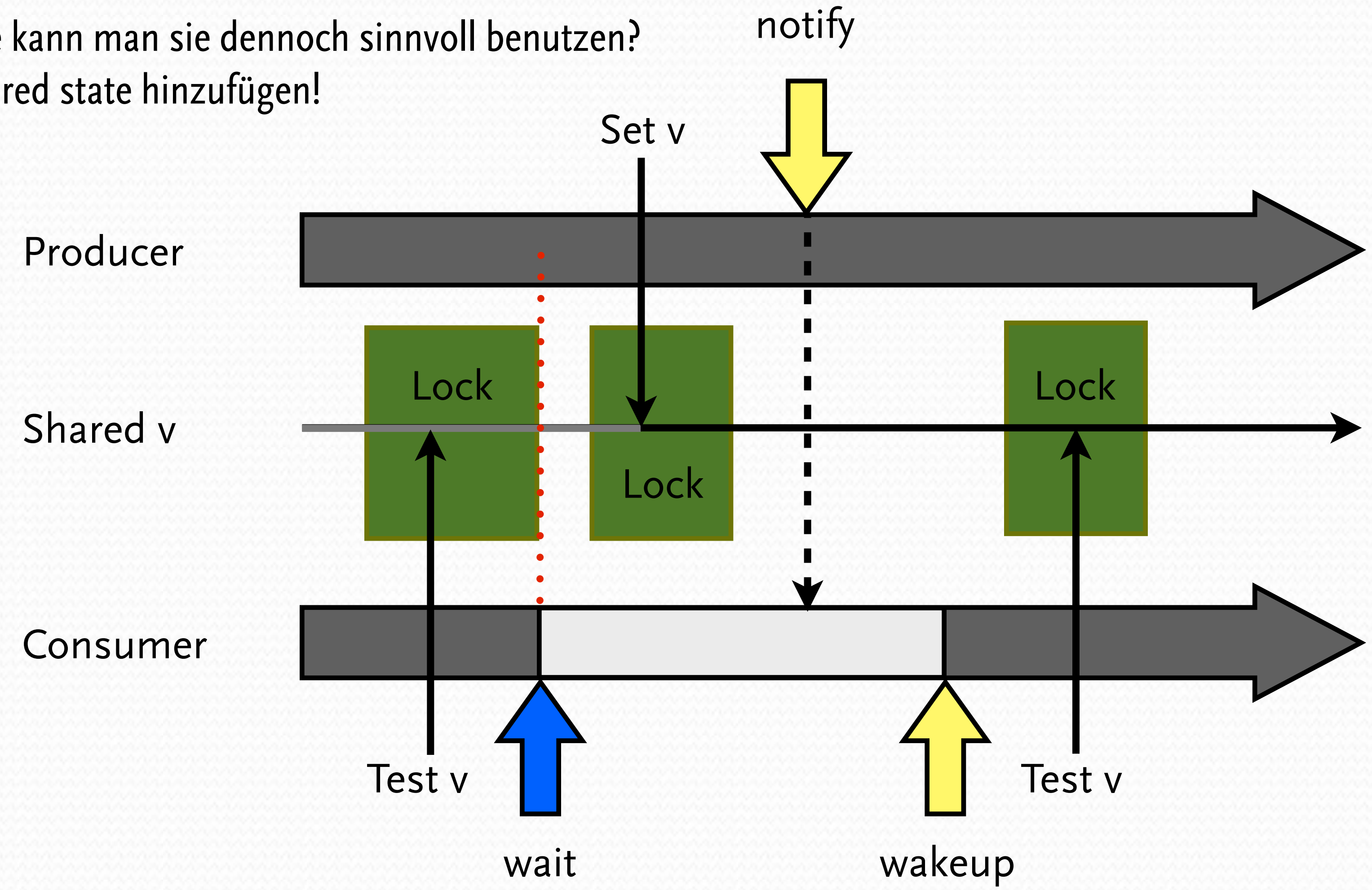
lck.unlock()

wait for notify

notify: relock before test

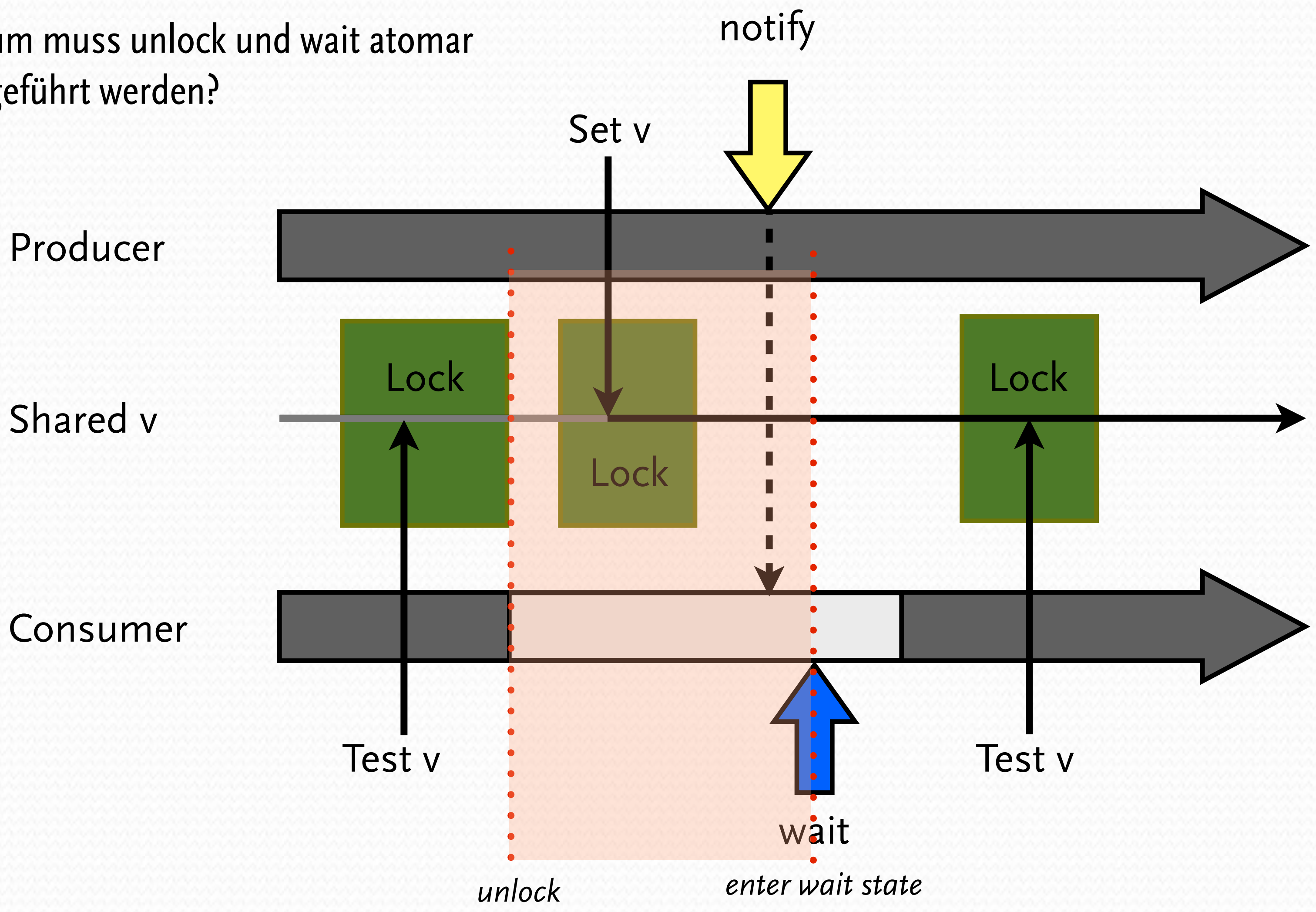
threads condition variables

Wie kann man sie dennoch sinnvoll benutzen?
Shared state hinzufügen!



threads condition variables

warum muss unlock und wait atomar ausgeführt werden?



threads condition variables

Shared Variable Protokoll: hiding the loop - looping inside of wait

Producer Thread

```
v = true;

cond.notify_one();
```

Consumer Thread

```
while (!v)

  cond.wait(?);
```

Locking Protokoll

```
{
 lock_guard lck(mtx);
 v = true;
}
cond.notify_one();
```

```
unique_lock lck(mtx);

cond.wait(lck,
  [&v]{return v;});
```

threads condition variables

ein praktisches Beispiel

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond; // #include <condition_variable>

void data_preparation_thread() {
  while(more_data_to_prepare()) {
    data_chunk const data = prepare_data(); // outside the lock !
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data); // outside the lock !
    if(is_last_chunk(data)) break;
  }
}
```

threads atomic types

einfachster Typ `std::atomic_flag`

- ‚nur ein Bit‘, muss initialisiert werden mittels

```
std::atomic_flag flag = ATOMIC_FLAG_INIT; // means: clear
```

- garantiert lock-frei, nur diese Operationen (keine nichtmodifizierende Abfrage möglich !)

`flag.clear([memory_order]);` und

```
bool was = flag.test_and_set([memory_order]); // set now
```

kann für minimalen spin-lock mutex verwendet werden

threads atomic types

```
// spin-lock mutex (user code)

class spinlock_mutex {
    std::atomic_flag flag;
public:
    spin_lock_mutex(): flag(ATOMIC_FLAG_INIT) {}

    void lock() {
        while(flag.test_and_set(std::memory_order_acquire));
    }

    void unlock() {
        flag.clear(std::memory_order_release);
    }
}

// very basic BUT enough for std::lock_guard<>
```

threads atomic types

std::atomic<bool>

```
   std::atomic<bool> b (true); // or false
   // std::atomic<bool> b = true; nicht erlaubt: atomic(const atomic&) = delete; aber
   std::atomic<bool> b = ATOMIC_VAR_INIT(true);


   b = false; // b.store(false)    returns bool !

   b.store(new_value, [memory_order]);

   bool old_value = b.exchange(new_value, memory_order);

   bool changed = b.compare_exchange_strong(expected,new_value,[memory_order]);

   // hat b den wert expected, so setze auf new_value und return true
   // sonst aktualisiere expected (&) auf aktuellen wert von b und return false

std::atomic<bool> b (true); bool expect = true;
if(b.compare_exchange_strong(expect, false)) ... // it was me, no one else
else ... // some other thread ahead, actual value of b unknown
```

threads atomic types

std::atomic<bool>

```
bool maybe_changed = b.compare_exchange_weak(expected,new_value,[memory_order]);
```

eine schwächere Form: bei return true: b war expected und neuer wert wurde auf new_value gesetzt

aber bei return false: wert von b wurde nicht verändert (u.U. obwohl b expected war, *spurious failure*)

mit einer zusätzlichen Schleife absichern:

```
bool expected = false;
extern std::atomic<bool> b; // set somewhere else
while (!b.compare_exchange_weak(expected, true) && !expected)
  // now i am sure to be the one who set b to true
```

Es ist *implementation defined*, ob atomic<bool> Operationen Lock-frei sind:

```
if (b.is_lock_free()) // ja
```

threads more atomic types

atomic type	template specialization
atomic_bool	std::atomic<bool>
atomic_[u|s]char	std::atomic<[unsigned|signed] char>
atomic_[u]int	std::atomic<[unsigned] int>
atomic_[u]short	std::atomic<[unsigned] short>
atomic_[u]long	std::atomic<[unsigned] long>
atomic_[u]llong	std::atomic<[unsigned] long long>
atomic_char16_t	std::atomic<char16_t>
atomic_char32_t	std::atomic<char32_t>
atomic_wchar_t	std::atomic<wchar_t>

Es ist *implementation defined*, ob atomic<*integral*> Operationen Lock-frei sind:

```
if (i.is_lock_free()) // ja
```

threads atomic pointer types

`std::atomic<T*>`

atomare Operationen:

```
    load, store, exchange, compare_exchange_weak, compare_exchange_strong
```

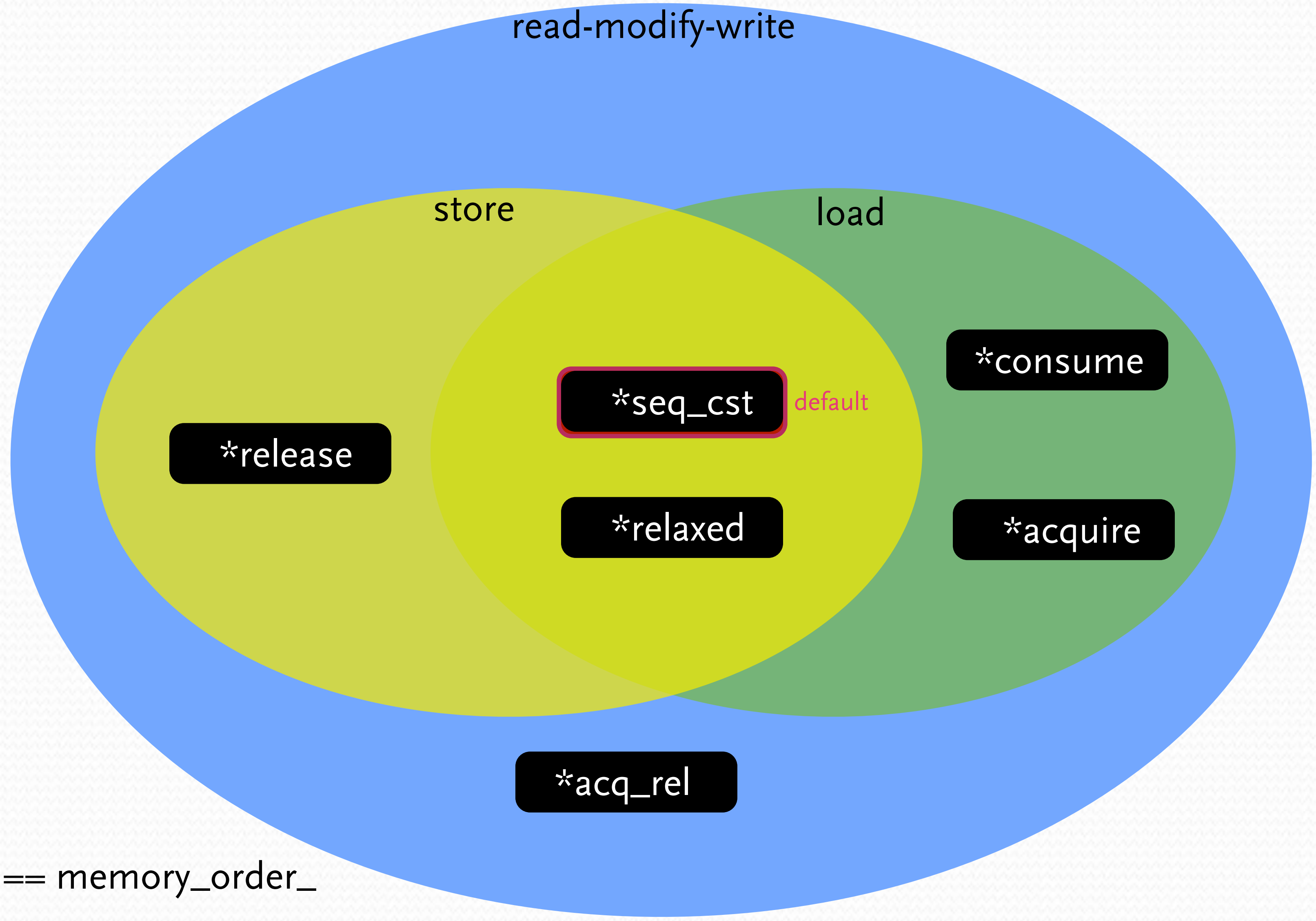
und zusätzliche atomare Zeigeroperationen:

```
fetch_add, fetch_sub, +=, -=, ++, -- (pre and postfix)

class Foo {};
Foo some_array [5];
std::atomic<Foo*> p(some_array);
Foo* x = p.fetch_add(2[, memory_order]); // x gives old value
assert(x == some_array);
assert(p.load([memory_order]) == &some_array[2])
x=(p-=1);
assert(x == &some_array[2]);
assert(p.load([memory_order]) == &some_array[1])
```

threads

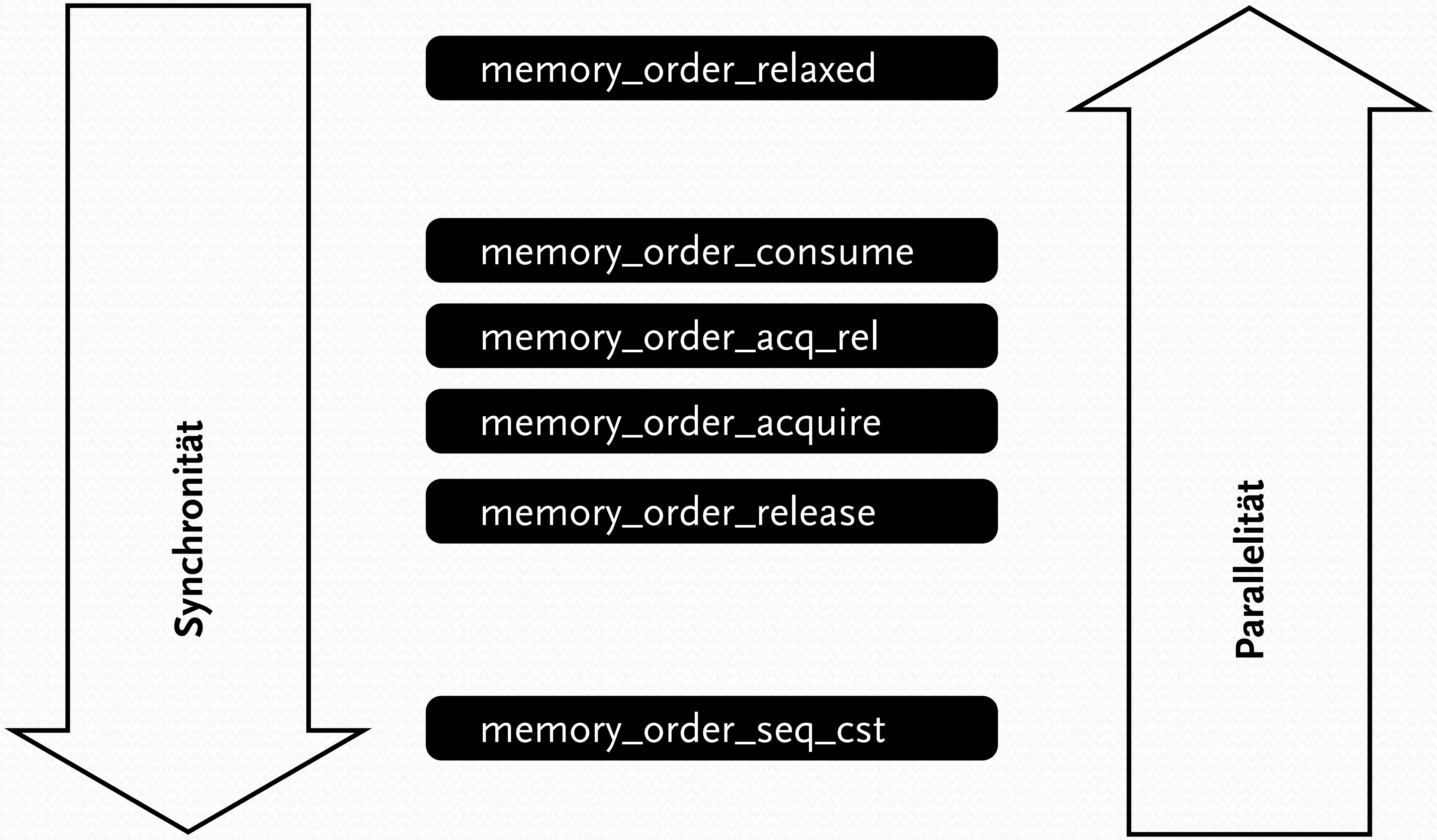
memory ordering of atomic operations



* == memory_order_

threads

memory ordering of atomic operations



threads memory ordering of atomic operations

Ob die Benutzung strikterer memory order auch zu stärkerer Kopplung von Threads führt (ggf. zusätzliche Warteoperationen um Garantien einzuhalten) ist vom Prozessor abhängig:

Anthony Williams: C++ Concurrency in Action (ISBN 13: 978-1-933988-77-1)

These distinct memory-ordering models can have varying costs on different CPU architectures. For example, on systems based on architectures with fine control over the visibility of operations by processors other than the one that made the change, additional synchronization instructions can be required for sequentially consistent ordering over acquire-release ordering or relaxed ordering and for acquire-release ordering over relaxed ordering. If these systems have many processors, these additional synchronization instructions may take a significant amount of time, thus reducing the overall performance of the system. On the other hand, **CPUs that use the x86 or x86-64 architectures (such as the Intel and AMD processors common in desktop PCs) don't require any additional instructions for acquire-release ordering beyond those necessary for ensuring atomicity, and even sequentially-consistent ordering doesn't require any special treatment for load operations, although there's a small additional cost on stores.**

The availability of the distinct memory-ordering models allows experts to take advantage of the increased performance of the more fine-grained ordering relationships where they're advantageous while allowing the use of the default sequentially consistent ordering (which is considerably easier to reason about than the others) for those cases that are less critical.

threads memory ordering of atomic operations

```
#include <thread>
#include <atomic>
#include <cassert>

std::atomic<bool> x = ATOMIC_VAR_INIT(false);
std::atomic<bool> y = ATOMIC_VAR_INIT(false);
std::atomic<int>  z = ATOMIC_VAR_INIT(0);

void write_x() {
    x.store(true, std::memory_order_seq_cst);
}

void write_y() {
    y.store(true, std::memory_order_seq_cst);
}

void read_x_then_y() {
    while (!x.load(std::memory_order_seq_cst));
    if (y.load(std::memory_order_seq_cst)) {   ++z;   }
}

void read_y_then_x() {
    while (!y.load(std::memory_order_seq_cst));
    if (x.load(std::memory_order_seq_cst)) {   ++z;   }
}
```

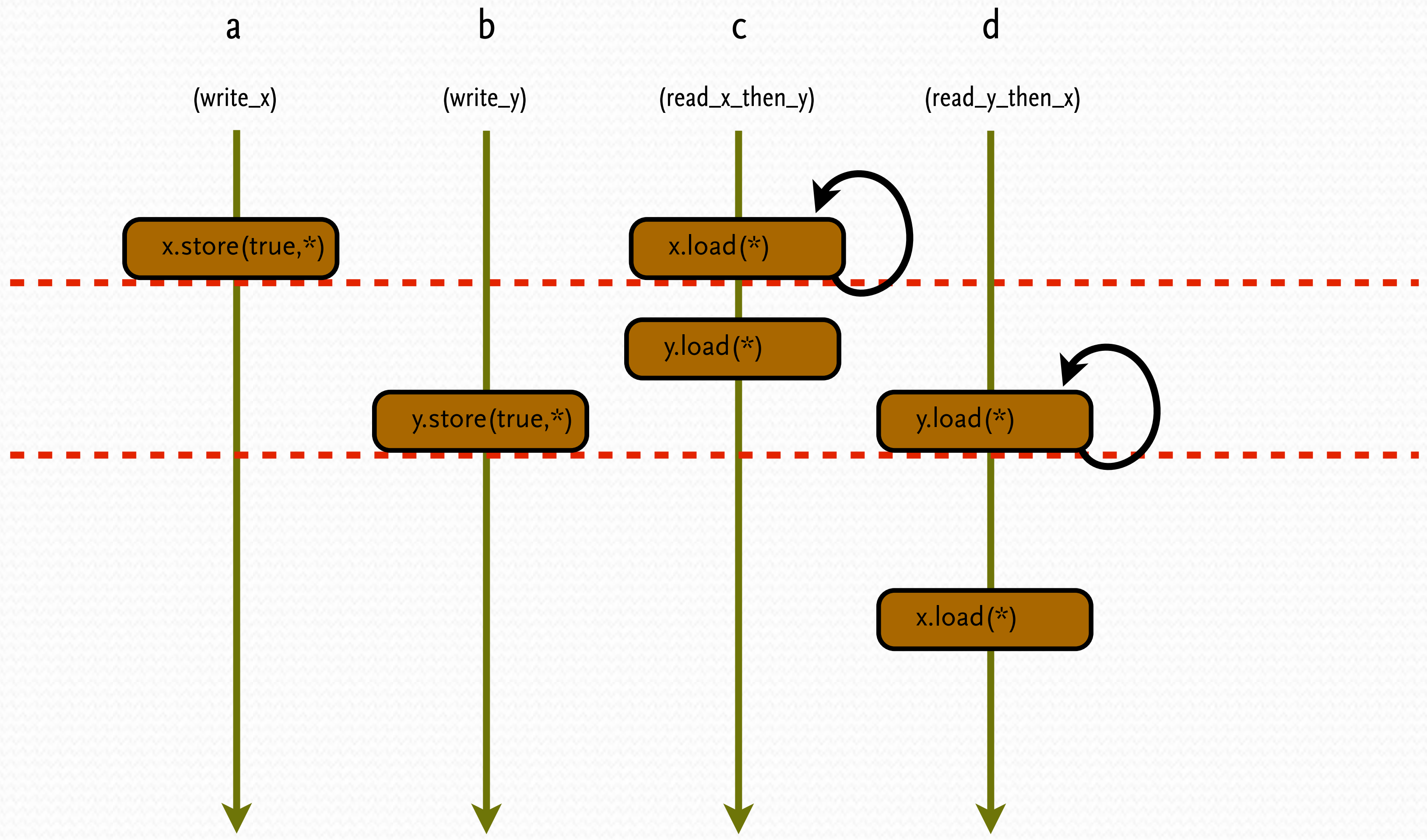
threads memory ordering of atomic operations

```
int main()
{
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join(); b.join(); c.join(); d.join();
    assert(z.load() != 0);  // will never happen
}

// mindestens einer der threads c und d macht ++z (u.U. auch beide)
```

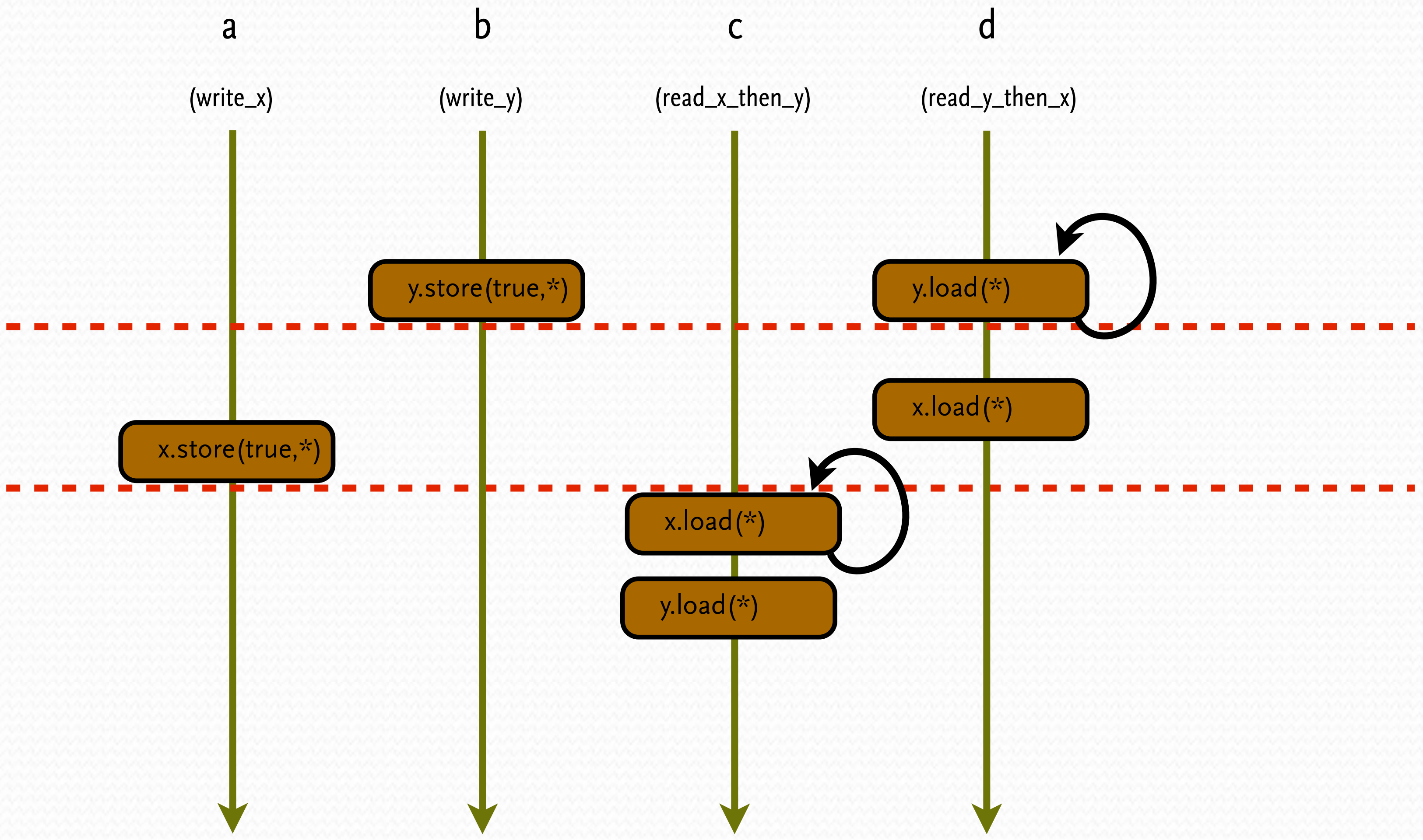
threads memory ordering of atomic operations

* - memory_order_seq_cst



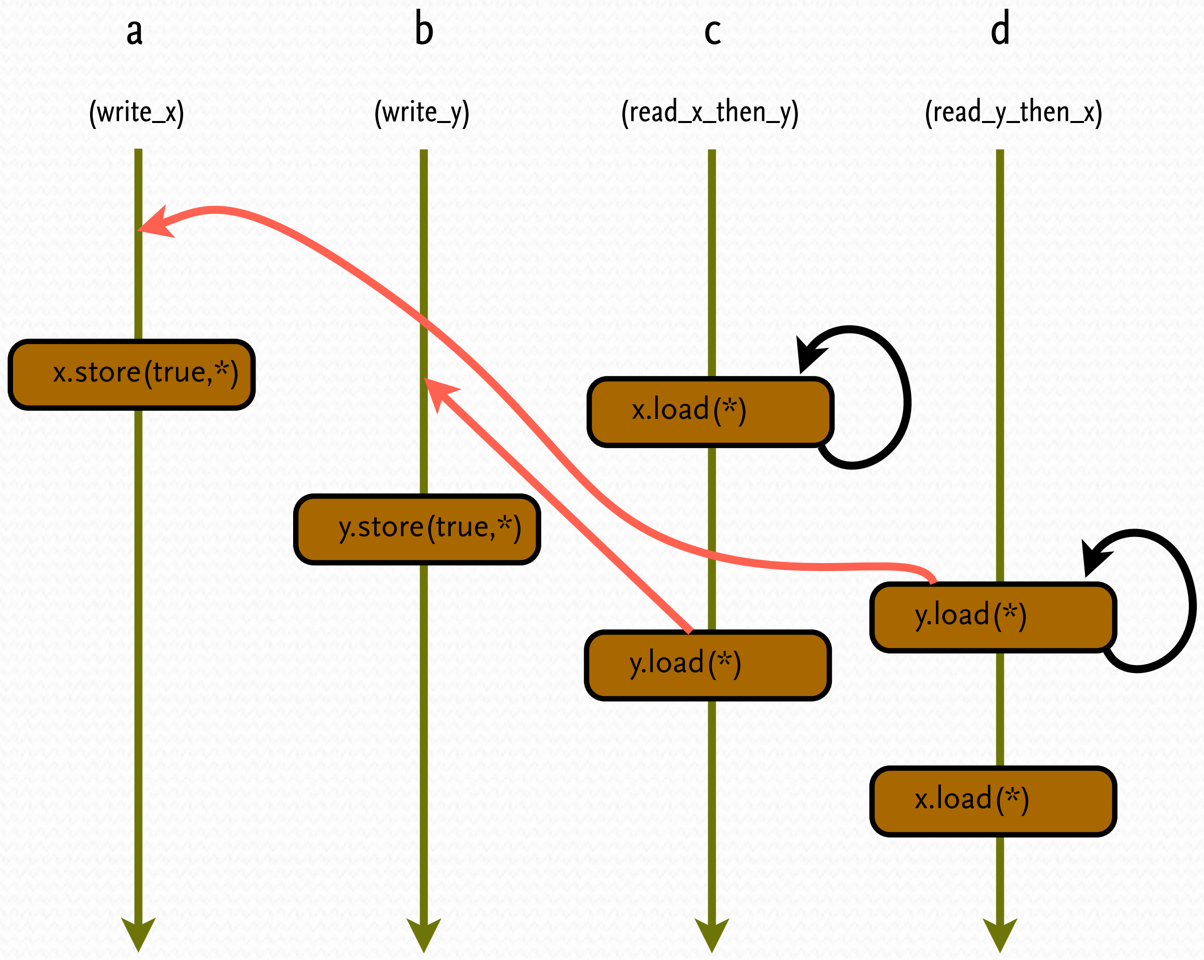
threads memory ordering of atomic operations

* - memory_order_seq_cst



threads memory ordering of atomic operations

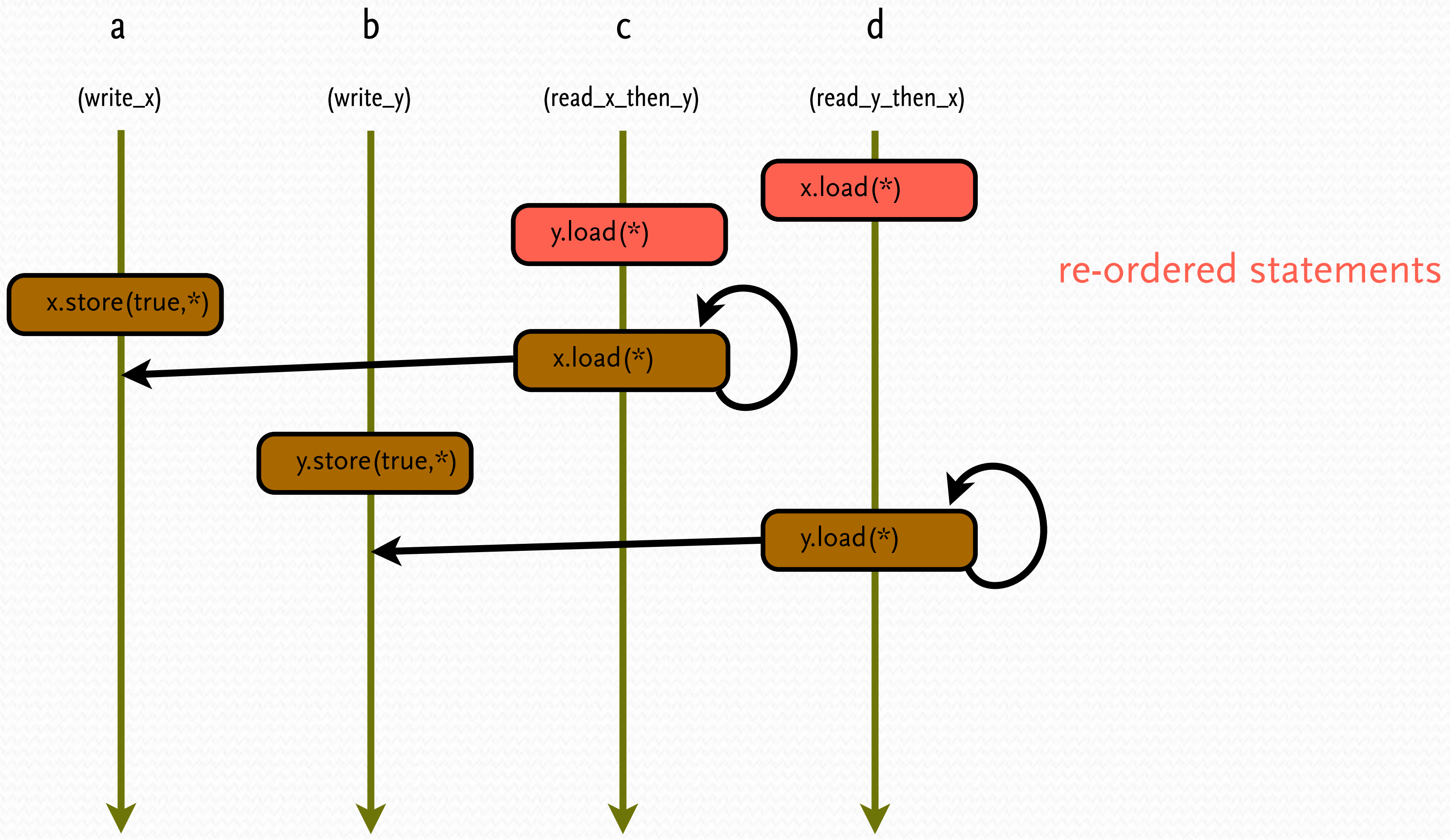
* - memory_order_relaxed



past values

3

threads memory ordering of atomic operations


* - memory_order_relaxed
a
b
c
d
(write_x)
(write_y)
(read_x_then_y)
(read_y_then_x)
x.load(*)
y.load(*)
re-ordered statements
x.store(true,*)
x.load(*)
y.store(true,*)
y.load(*)