Programmation Parallèle (PRPA)

E. Carlinet {edwin.carlinet@lrde.epita.fr}

2020

EPITA Research & Development Laboratory (LRDE)

Agenda

Memory ordering

The C++ memory model

Syncing and ordering

Atomics and mutexes

C++ atomics

C++ Relaxed Memory Orders

Agenda

Agenda

1. Introduction to parallelism
2. Instruction and data-level parallelism
3. Thread level parallism
4. Parallel Design Patterns (with TBB)
5. C++ Memory model
6. Data structure for concurrent programming

Memory ordering

Remainder

```
unsigned sum = 0;

void partial_sum(unsigned tab[], std::size_t n)
{
  for (std::size_t i = 0; i < n; ++i)
    sum += tab[i];
}

void sum(unsigned tab[], std::size_t n)
{
  auto _1 = std::async(partial_sum, tab, n/2);
  auto _2 = std::async(partial_sum, tab + n/2, n - n/2);
}
```

This code is not valid:

- we have a data race

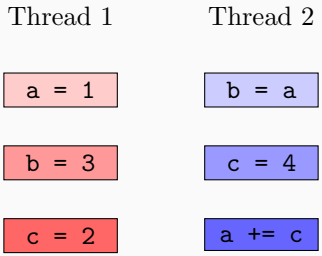
Remainder

An optimizing compiler can rewrite:

```
void partial_sum(unsigned tab[], std::size_t n)
{
  unsigned tmp = sum;
  for (std::size_t i = 0; i < n; ++i)
    tmp += tab[i];
  tmp = sum;
}
```

Instructions ordering

`a = 0; b = 0; c = 0` Suppose we have *atomic* instructions:



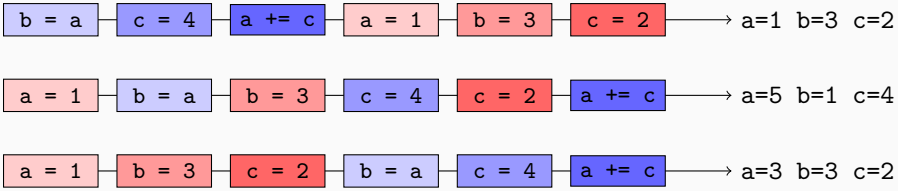
What are the possibles values for a,b,c after execution ?

Instructions ordering

`a = 0; b = 0; c = 0` Suppose we have *atomic* instructions:

Thread 1	Thread 2
`a = 1`	`b = a`
`b = 3`	`c = 4`
`c = 2`	`a += c`

What are the possibles values for a,b,c after execution ?



And much more…

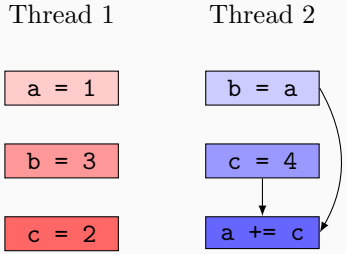
How many interleavings for 2 threads of N and M instructions ?

How many interleavings for 2 threads of N and M instructions ?

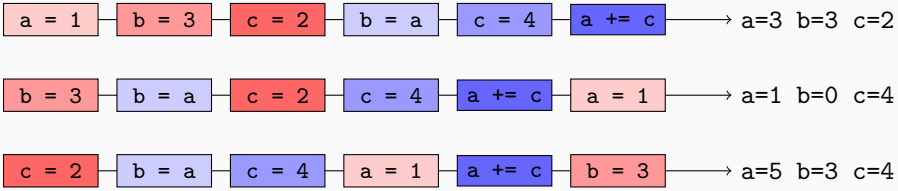
```
C(N,M) = C(N-1,M) + C(N,M-1)
C(0,M) = 1
C(N,0) = 1
```

For N=3 and M=3: 20 orderings

But what if we could switch some instructions without changing per-thread semantics:



Note that `a += c` has dependancy on `c = 4` and `b = a`



We have 51 reorderings !

The Truth about your program

- Your program:

```
if (condition) my_global = 4;
int k = my_global;
do_stuff_with_k
```

The Truth about your program

- Your program:

```
if (condition) my_global = 4;
int k = my_global;
do_stuff_with_k
```

- What the compiler might say:

```
int k = condition ? 4 : my_global;
do_stuff_with_k
if (condition) my_global = 4;
```

The Truth about your program

- Your program:

```
if (condition) my_global = 4;
int k = my_global;
do_stuff_with_k
```

- What the compiler might say:

```
int k = condition ? 4 : my_global;
do_stuff_with_k
if (condition) my_global = 4;
```

- What the processor might say:

```
k = 4
do_stuff_with_k
my_global = 4;
if (condition) return;
k = my_global;
do_stuff_with_k
```

The Truth about your program

DSE (Dead store elimination)

```
x = 1;
y = "universe";
x = 2;
```

```
y = "universe";
x = 2;
```

Memory-to-register

```
for (int i = 0; i < len; ++i)
    z += tab[i];
```

```
int r1 = z;
for (int i = 0; i < len; ++i)
    r1 += tab[i];
z = r1;
```

The Truth about your program

The compiler knows:

- Operations in **one thread** with data dependancies
- Pointer aliasing issues

And they better not execute the *stupid* code you have writen. Your code is slow !

- Compiler better optimize and reorder
- Processors can do speculative branching and out-of-order exec

- Program executed $\neq$ program you wrote

The Truth about your program

The compiler does *not* know:

- How memory can be affected by other thread

- We have to tell the compiler about it
- We have to limit the way things are reordered to get consistent results

What we want (SC)

Sequential Consistency (Lamport 79):

- the result of any execution is the same as if the operations of all the processors were executed in some sequential order
- **and the operations of each individual processor appear in this sequence in the order specified by its program**

- Execute what I wrote !

Solutions

1. Follow the C++ memory model.
2. How limit the way things are reordered to get consistent results ?

- Atomic operations for types
- Critical sections for block of code
- Memory fences

The C++ memory model

The C++ Memory model

You agree

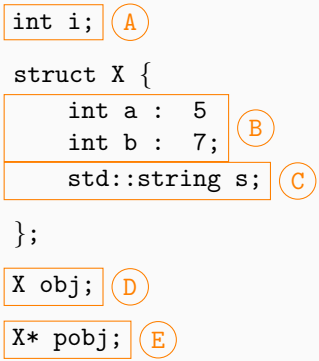
To correctly synchronise your code

You get

The *illusion* of a sequential consistent execution

The C++ Memory model

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



The C++ Memory model

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

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

```
int i;                  A

struct X {
    int a :  5
    int b :  7;         B
    std::string s;      C
};

X obj;                  D

X* pobj;                E
```

Is this program ok ?

Thread 1	Thread2
`i = i + 2`	`obj.s.push_back('a');`
`obj.a = i`	`pobj = &obj;`
`X x = obj`	`pobj->b = 5;`

```
int i;                  (A)

struct X {
    int a : 5
    int b : 7;          (B)
    std::string s;      (C)
};

X obj;                  (D)

X* pobj;                (E)
```

Thread 1	Thread2
`i = i + 2`	`obj.s.push_back('a');` (4)
`obj.a = i` (1)	`pobj = &obj;`
`X x = obj` (2)	`pobj->b = 5;` (3)

- (B) is writen by Thread 1 (1) and Thread 2 (3)
- (C) is read by Thread 1 (2)
- (C) is writen by Thread 2 (4)

The C++ Memory model

```
int x = 0, y = 0;
```

```
if (x == 1)
    y = 1;
```

```
if (y == 1)
    x = 1;
```

Is there a data race?

The C++ Memory model

```
int x = 0, y = 0;
```

```
if (x == 1)
    y = 1;
```

```
if (y == 1)
    x = 1;
```

Is there a data race?

No ! (Whichever thread starts first, there will be no write)

Modification order Every memory location has a **modification order** composed of all the writes to that object from all threads in the program

No data race = Single modification order for all threads (that can vary between two program executions).

Syncing and ordering

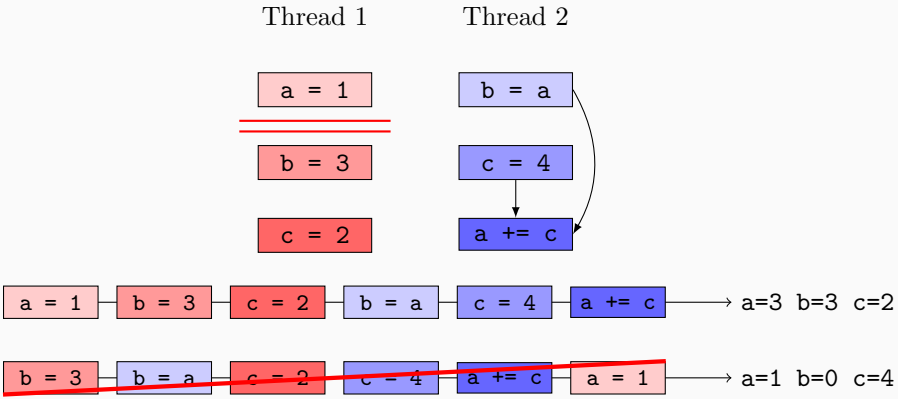
Syncing and ordering

How limit the way things are reordered to get consistent results?

- Memory fences
- Atomic operations for types
- Critical sections for block of code

Fences for memory ordering

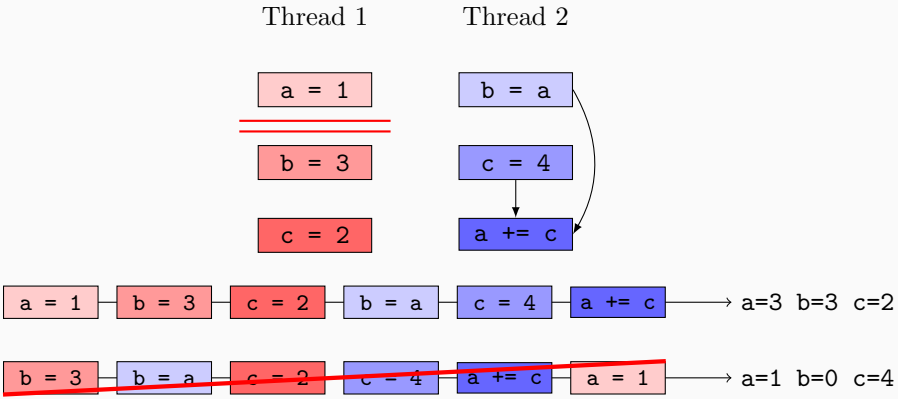
First usage: limit the number of reordering (**Memory ordering**)



Possibles values for `b` after executions of threads:

Fences for memory ordering

First usage: limit the number of reordering (**Memory ordering**)



Possibles values for `b` after executions of threads:

0, 1, 3

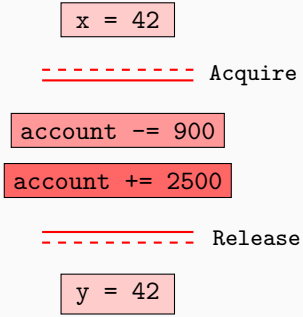
Fences for memory ordering

General idea:

- Code cannot **move-out**
- But code can **move in**

```
int account = 0, x = 0, y = 0;
```

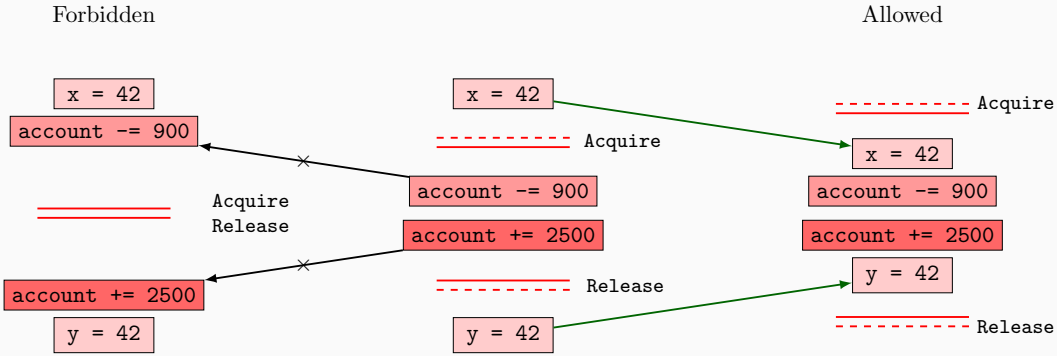
```
x = 42;
std::atomic_thread_fence(std::memory_order_acquire);
account -= 900;
account += 2500;
std::atomic_thread_fence(std::memory_order_release);
y = 42;
```



Fences for memory ordering

General idea:

- Code cannot **move-out**
- But code can **move in**



Fences for memory ordering

```
x = 42;
account -= 900;
account += 2500;
x += 42;
```

```
add dword ptr [rip + account], 1600
mov dword ptr [rip + x], 84
```

Fences for memory ordering

```
x = 42;
account -= 900;
account += 2500;
x += 42;
```

```
add dword ptr [rip + account], 1600
mov dword ptr [rip + x], 84
```

```
x = 42;
account -= 900;
account += 2500;
std::atomic_thread_fence(release);
x += 42;
```

```
add dword ptr [rip + account], 1600
mov dword ptr [rip + x], 42
mov dword ptr [rip + x], 84
```

Fences for memory ordering

```
x = 42;
account -= 900;
account += 2500;
x += 42;
```

```
add dword ptr [rip + account], 1600
mov dword ptr [rip + x], 84
```

```
x = 42;
account -= 900;
account += 2500;
std::atomic_thread_fence(release);
x += 42;
```

```
add dword ptr [rip + account], 1600
mov dword ptr [rip + x], 42
mov dword ptr [rip + x], 84
```

```
x = 42;
std::atomic_thread_fence(acquire);
account -= 900;
account += 2500;
x += 42;
```

```
mov dword ptr [rip + x], 42
add dword ptr [rip + account], 1600
add dword ptr [rip + x], 42
```

Syncing with fences: inside a thread

- Fences allow to limit instruction redordering (by CPU or compiler)
- They enforce instructions ordering:

Acquire operation: No read or writes can be redordered before this operation
Release operation: No read or write can be reordered after this operation

---

They provide (in-thread) **happens-before** relationship.

Syncing with threads: for other threads

- Fences allow to **commit** to others. Side-effects become visible

---

Release operation:

- Make visible **previous** writes (push)

Acquire operation:

- See what has been commited (pull)

- They provide a **synchronise with** relationship
- **synchronise-with** = *inter-thread* **happens-before**

Fences for synchronization (the bad)

Warning Fences do not prevent data races (concurrent read/write write/write)

```
bool has_been_payed = false;
int account = 0;
```

```
account += 2500;
std::atomic_thread_fence(release);
has_been_payed = true;
```

```
bool ok = has_been_payed;
std::atomic_thread_fence(acquire);
if (ok)
    assert(account > 0);
```

Is it OK?

Fences for synchronization (the bad)

Warning Fences do not prevent data races (concurrent read/write write/write)

```
bool has_been_payed = false;
int account = 0;
```

```
account += 2500;
std::atomic_thread_fence(release);
has_been_payed = true;
```

```
bool ok = has_been_payed;
std::atomic_thread_fence(acquire);
if (ok)
    assert(account > 0);
```

Is it OK?

Data race on `has_been_payed` !

- In T1: *Write to* `account` *Happens-before Write to* `has_been_payed`
- In T2: *Read to* `account` conditional to *Write to* `has_been_payed`
- *Write to* `account` *Happens-before Read to* `account` (by *Synchronize-with* of fences)

Atomics and mutexes

Mutexes

	Fences	Mutexes
Synchronization	✓	✓
Memory ordering	✓	✓
Transactional model	✗	✓

Transaction

- Atomic: all or nothing
- Consistent: bring one *consistent* state to another *consistent* state
- Independant: Correct if other transactions appear in the same time

With *mutexes*, **Independance** = **Mutual exclusion** (two threads cannot execute a critical section).

```
int account = 0, x = 0, y = 0;
```

```
x = 42;
m.lock();   // Acquire operation
account -= 900;
account += 2500;
m.unlock(); // Release operation
y = 42;
```

```
x = 42;
m.lock();   // Acquire operation
account -= 900;
account += 2500;
m.unlock(); // Release operation
y = 42;
```

Is it OK ?

```
int account = 0, x = 0, y = 0;
```

```
x = 42;
m.lock();   // Acquire operation
account -= 900;
account += 2500;
m.unlock(); // Release operation
y = 42;
```

```
x = 42;
m.lock();   // Acquire operation
account -= 900;
account += 2500;
m.unlock(); // Release operation
y = 42;
```

Is it OK ?

Still not !

Concurrent write to x and y (but “benign” as writing the same int value).

Mutexes & Synchronization

```
int account = 0, x = 0, y = 0;
```

```
m.lock();   // Acquire operation
x = 42;
account -= 900;
account += 2500;
y = 42;
m.unlock(); // Release operation
```

```
m.lock();   // Acquire operation
x = 42;
account -= 900;
account += 2500;
y = 42;
m.unlock(); // Release operation
```

Mutexes & Synchronization

```
int x = 0, y = 0;
```

```
x++;
m.lock();
y++;
m.unlock();
```

```
m.lock();
if (y++)
    x++;
m.unlock();
```

OK ? Which ouput ?

Mutexes & Synchronization

```
int x = 0, y = 0;
```

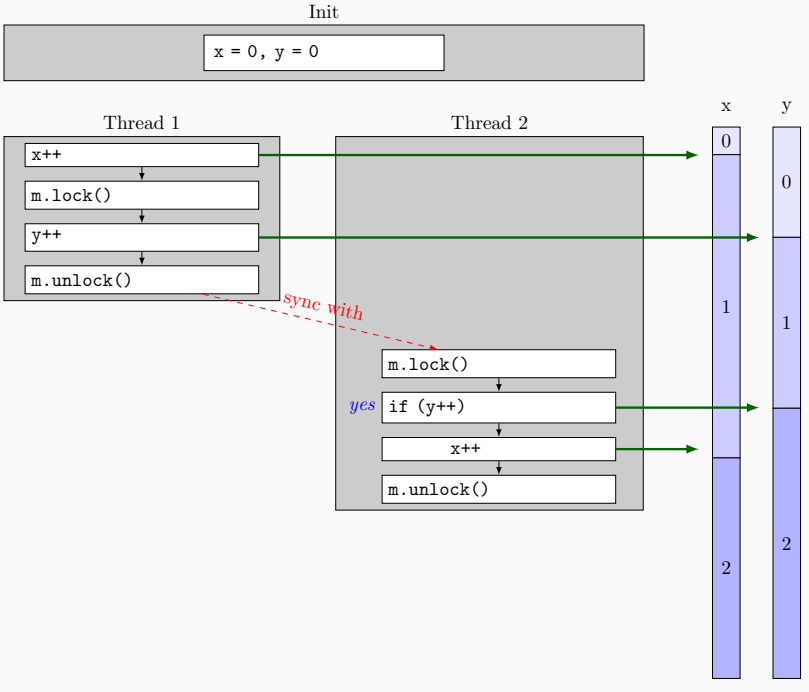
```
x++;
m.lock();
y++;
m.unlock();
```

```
m.lock();
if (y++)
    x++;
m.unlock();
```

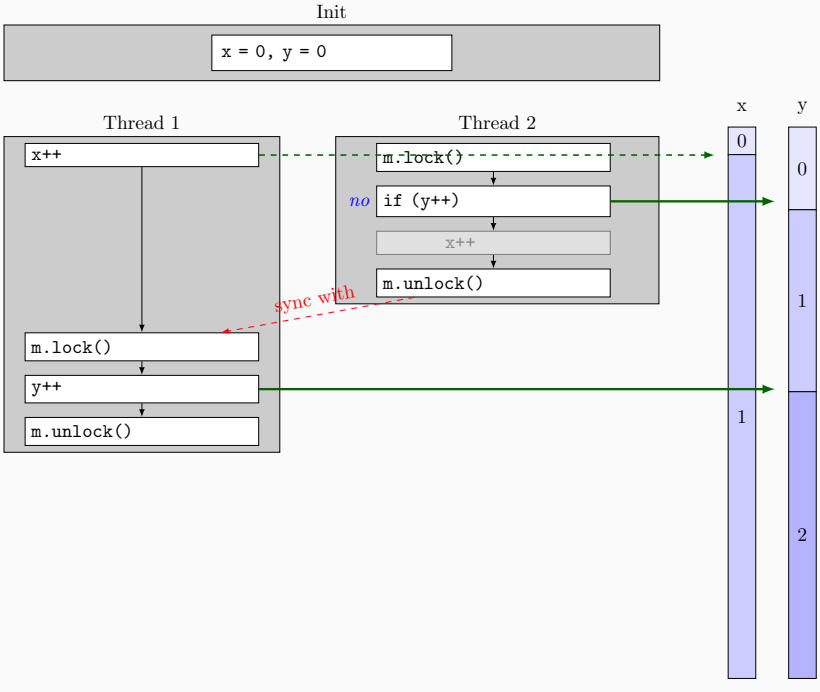
OK ? Which ouput ?

- `assert(y == 2)`
- `assert(x == 1 || x == 2)`

First case `x == 2`



First case `x == 1`



Exercice: make it correct (with minimum locking)

```
bool has_been_payed = false;
int account = 0;
```

```
account += 2500;
std::atomic_thread_fence(release);
has_been_payed = true;
```

```
bool ok = has_been_payed;
std::atomic_thread_fence(acquire);
if (ok)
    assert(account > 0);
```

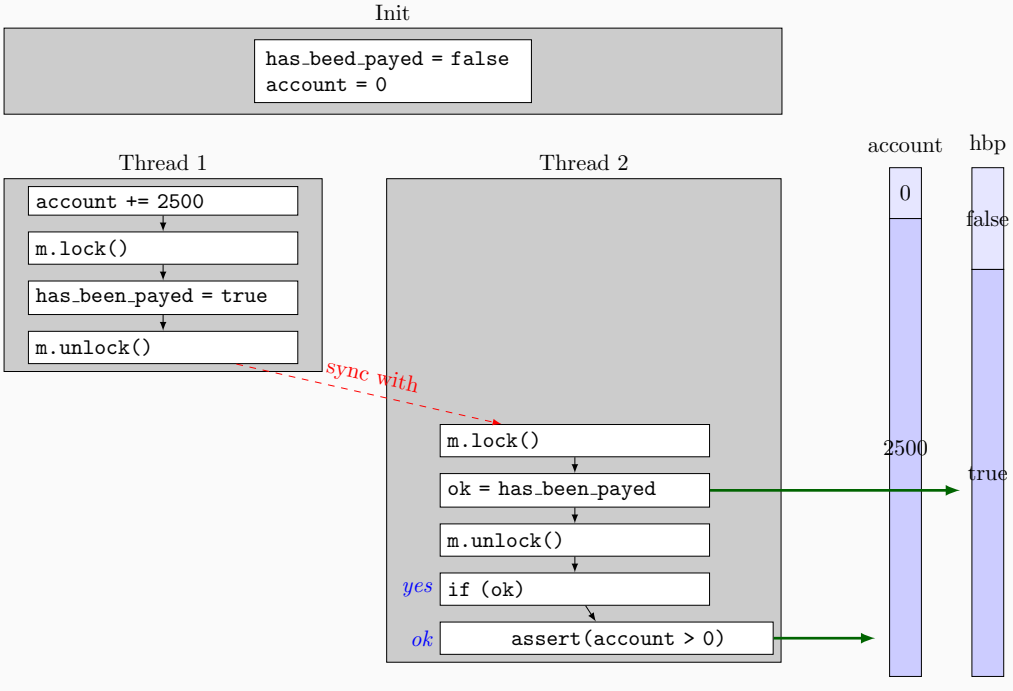
```
bool has_been_payed = false;
int account = 0;
```

```
account += 2500;
m.lock();
has_been_payed = true;
m.unlock();
```

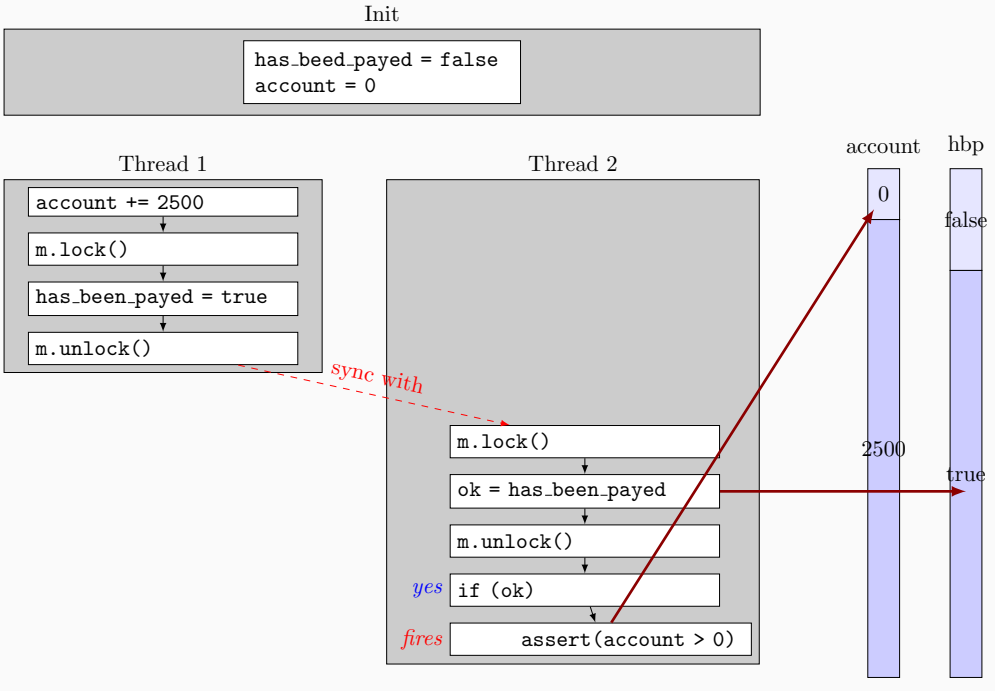
```
bool ok;
m.lock();
ok = has_been_payed;
m.unlock();
if (ok)
    assert(account > 0);
```

Why does it work ?

Possible exec 1



Not possible



Mutexes & C++ RAII Note

```
std::mutex m;

void foo()
{
 m.lock();
 ...
 m.unlock();
}
```

What do you think about:

- exceptions
- programming errors

Mutexes & C++ RAII Note

```
std::mutex m;

void foo()
{
 m.lock();
 ...
 m.unlock();
}
```

This is safer written as:

```
{
 std::lock_guard l(m); // Acquire
 ...
 // Release at destruction
}
```

What do you think about:

- exceptions
- programming errors

C++ atomics

C++ Atomics

	Fences	Mutexes	Atomics
Synchronization	✓	✓	✓
Memory ordering	✓	✓	✓
Transactional model	✗	✓(Code block)	✓(Single op)

Transaction:

- Atomic: all or nothing
- Consistent: bring one *consistent* state to another *consistent* state
- Independant: Correct if other transactions appear in the same time

C++ Atomics

	Fences	Mutexes	Atomics
Synchronization	✓	✓	✓
Memory ordering	✓	✓	✓
Transactional model	✗	✓(Code block)	✓(Single op)

Transaction:

- Atomic: all or nothing
- Consistent: bring one *consistent* state to another *consistent* state
- Independant: Correct if other transactions appear in the same time

---

- `std::atomic<T>` is the **only** way to get **atomic** operations in C++
- Most `std::atomic<P>` (with `P` a primitive type) are lock-free
- `std::mutex` and spin lock use atomics behind

Operations on C++ Atomics

Operation	atomic_flag	atomic<bool>	atomic<T*>	atomic<int>
test_and_set	✓			
clear	✓			
is_lock_free		✓	✓	✓
load / operator T		✓	✓	✓
store / operator=		✓	✓	✓
exchange		✓	✓	✓
compare_exchange_strong/weak		✓	✓	✓
fetch_add/sub += -= ++ --			✓	✓
fetch_or/and/xor				✓

Atomic Traps

Some of these are not like the others:

- `std::atomic<int> x{0}`
- `++x`
- `x++`
- `x += 1`
- `x |= 2`
- `x *= 2`
- `int y = x * 2`
- `x = y + 1`
- `x = x + 1`
- `x = x * 2`

Atomic Traps

Some of these are not like the others:

- `std::atomic<int> x{0}`
- `++x`
- `x++`
- `x += 1`
- `x |= 2`
- ~~`x *= 2`~~ (does not compile)
- `int y = x * 2`
- `x = y + 1`
- ~~`x = x + 1`~~ (not atomic +)
- ~~`x = x * 2`~~ (not atomic *)

Exercice: make it correct (with atomic)

```
bool has_been_payed = false;
int account = 0;
```

```
account += 2500;
std::atomic_thread_fence(release);
has_been_payed = true;
```

```
bool ok = has_been_payed;
std::atomic_thread_fence(acquire);
if (ok)
    assert(account > 0);
```

C++ Atomics with `atomic<bool>`

```
std::atomic<bool> has_been_payed(false);
int account = 0;
```

```
account += 2500;
// Release operation
has_been_payed.store(true);
```

```
// Acquire operation
if (has_been_payed.load())
    assert(account > 0);
```

C++ Atomics: with `atomic_flag`

```
std::atomic_flag has_been_payed = ATOMIC_FLAG_INIT;
int account = 0;
```

```
account += 2500;

// Acquire/Release operation
has_been_payed.test_and_set();
```

```
// Acquire/Release operation
if (has_been_payed.test_and_set())
    assert(account > 0);
```

Atomic exchange / compare_exchange

exchange(y)

Set and return the old value

```
T exchange(T new)
{
  T old = load();
  store(new);
  return old;
}
```

Atomic exchange / compare_exchange

compare_exchange(expected, desired)

Test if the current value is `expected`, if so set the new value with `desired` and returns `true`, else set `expected` to the current value, and returns `false`.

```
bool compare_exchange(T& expected, T desired)
{
  T current = load();
  if (current == expected) {
    store(desired);
    return true;
  } else {
    expected = current;
    return false;
  }
}
```

The CAS Loop

The most important loop with atomics:

```
while (flag.test_and_set())
    ;
```

or

```
while (value.compare_exchange_weak(expected, desired))
    ;
```

Note:

- `compare_exchange_weak` if in a loop (weak can fail *spuriously*)
- `compare_exchange_strong` if not a loop

The CAS Loop

Exercice: how do you implement a spinlock mutex with a CAS loop?

```
class spinlock_mutex
{
  std::atomic_flag m_flag = ATOMIC_FLAG_INIT;
public:
  void lock() {FIXME}
  void unlock() {FIXME}
};
```

The CAS loop

```
class spinlock_mutex
{
    std::atomic_flag m_flag = ATOMIC_FLAG_INIT;
public:
  void lock()
  {
    while (m_flag.test_and_set())
     ;
  }
  void unlock() { m_flag.clear(); }
};
```

- `lock`
  While the ticket is taken { Mark the ticket as taken }
  Mark the ticket as taken

- `unlock`
  Give back the ticket

C++ Relaxed Memory Orders

Why relaxed memory orders?

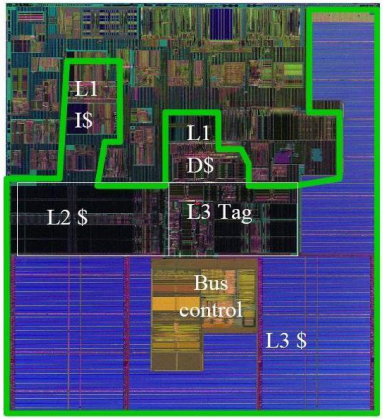
Remember course 1:

Strategy / Technique [1]	Affect your code
Parallelize (leverage compute power)	
Pipeline out-of-order	✓
Hardware (hyper) thread	
Cache (leverage capacity)	
Instruction cache	
Data cache	✓
Other buffering (e.g. store buffer)	✓
Speculate (leverage bandwidth/compute)	
Branch prediction	
Optimistic execution	
Prefetch	

[1] Herb Stuter - Atomic Weapons 2012

Why relaxed memory orders?

Remember course 1



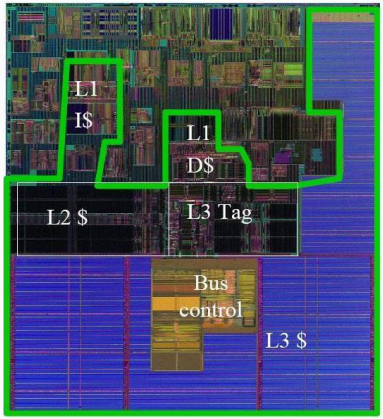
Original Itanium 2[a]:

- 211Mt *85% for cache*
- 1% of die to compute
- 99% to move/store data

[a]David Patterson, UC Berkeley, HPEC keynote, Oct 2004

Why relaxed memory orders?

Remember course 1



Original Itanium 2[a]:

- 211Mt *85% for cache*
- 1% of die to compute
- 99% to move/store data

[a]David Patterson, UC Berkeley, HPEC keynote, Oct 2004

Your computer is working hard to reorder & parallelize work, so:

- do not impose unnecessary ordering
- do not impose unnecessary synchronization

Why relaxed memory orders?

Different architectures = different memory models = different costs

Why relaxed memory orders?

	Ordinary Load	Ordinary Store	Atomic SC Load	Atomic SC Store	CAS
x86/x64	`mov`	`mov`	`mov`	`xchg`	`cmpxchg`

- Cheap SC load
- Expensive SC write (xchg = mov + mfence)

Reads are not reordered with any reads.
Writes are not reordered with any writes [some exceptions]

Writes are not reordered with older reads.
Reads may be reordered with older writes [different locations] .
Reads & writes not reordered with locked instructions [like xchg; . . .] .
Reads cannot pass earlier LFENCE and MFENCE.
Writes cannot pass earlier LFENCE, SFENCE, and MFENCE.
LFENCE cannot pass earlier reads.
SFENCE cannot pass earlier writes.
MFENCE cannot pass earlier reads or writes.

Why relaxed memory orders?

Different architectures = Different costs

	Ordinary Load	Ordinary Store	Atomic SC Load	Atomic SC Store	CAS
x86/x64	`mov`	`mov`	`mov`	`xchg`	`cmpxchg`
IA64	`ld`	`st`	`ld.acq`	`sr.rel; mf`	`cmpchg.rel; mf`
POWER	`ld`	`st`	`sync`; ld; cmp; bc; isync	`sync`; `st`	`sync` _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit:
ARM v7	`ldr`	`str`	ldr; `dmb`	`dmb`; str; `dmb`	`dmb`; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb

SC is **too** strong = We pay for something we **may** not need

SC is the *only* memory model for many languages (JAVA/ C#)
SC is the *default* behavior with C++

SC-Acquire-Release

So far, we have seen **Sequential Consitency** Acquire-Release.

Easy:

- Consistent with a single view of the world (as if a thread does all the work)
- **SC** = total orderings of the operations

Total ordering means:

- All threads agree on the modification order (not restricted to SC)
- All threads agree on the same memory state
- Get an *up-to-date* value for `load()`

SC-Example

Note:

```
auto seq_cst = std::memory_order_seq_cst;
```

```
std::atomic<bool> x = false, y = false;
```

```
x.store(true, seq_cst)
```

```
y.store(true, seq_cst)
```

```
while (!x.load(seq_cst))
  ;
if (!y.load(seq_cst))
  printf("x before y");
```

```
while (!y.load(seq_cst))
  ;
if (!x.load(seq_cst))
  printf("y before x");
```

Q: What are the possible outputs?

SC-Example

Note:

```
auto seq_cst = std::memory_order_seq_cst;
```

```
std::atomic<bool> x = false, y = false;
```

```
x.store(true, seq_cst)
```

```
y.store(true, seq_cst)
```

```
while (!x.load(seq_cst))
  ;
if (!y.load(seq_cst))
  printf("x before y");
```

```
while (!y.load(seq_cst))
  ;
if (!x.load(seq_cst))
  printf("y before x");
```

Q: What are the possible outputs?

- Nothing
- `x before y`
- `y before x`
- ~~But not both~~

Non-SC memory order

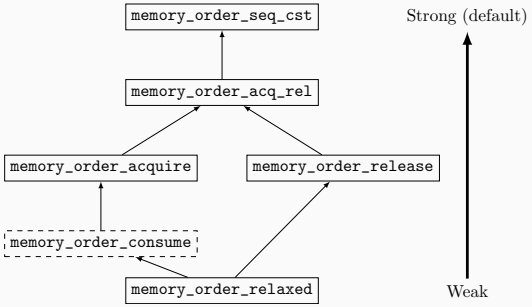
Break your way of thinking. It is *not only* about:

- Instructions reordering by hardwares (OoO…)
- Instructions/code reordering by compilers

With a non-SC memory order, 2 threads executing the same code:

- Agree about the modification order of variables
- Can disagree about the ordering of events (can see an old memory state)

Non-SC memory order



Relaxed	no operation orders memory (no syncing)
Release	store = release on the memory location
Acquire	load = acquire on the memory location
Consume	load = consume on the memory location

- Only affect memory ordering (and what becomes visible)!
- Operations are still atomics !

Relaxed Memory Order

- No operation orders memory (= no syncing = no *inter-thread* **happens-before**)
- Only atomicity guarantied (data-race free)

Relaxed Memory Order

```
auto relaxed = std::memory_order_relaxed;
std::atomic<int> account = 0;
std::atomic<bool> has_been_payed = false;
```

```
account.fetch_add(2500, relaxed);
has_been_payed.store(true, relaxed);
```

```
if (has_been_payed.load(relaxed))
   assert(account.load(relaxed) > 0)
```

Data race? Does the assert raise?

Relaxed Memory Order

```
auto relaxed = std::memory_order_relaxed;
std::atomic<int> account = 0;
std::atomic<bool> has_been_payed = false;
```

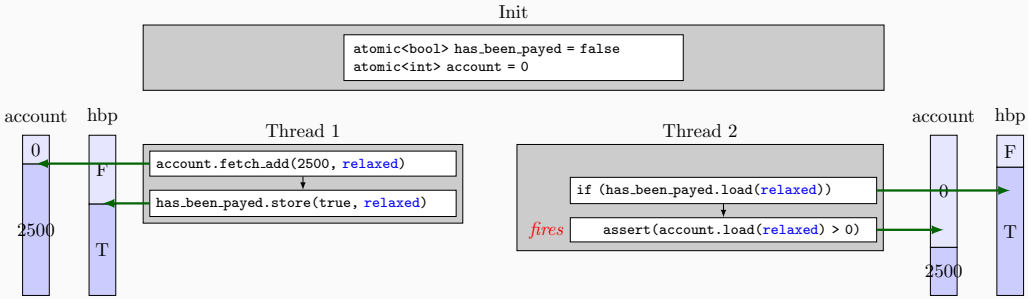
```
account.fetch_add(2500, relaxed);
has_been_payed.store(true, relaxed);
```

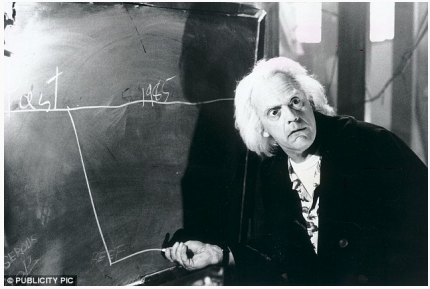
```
if (has_been_payed.load(relaxed))
   assert(account.load(relaxed) > 0)
```

Data race? Does the assert raise?

- Data race? No (all atomic operations)
- OK? No (we can see an old value because no sync)

Relaxed Memory Order





We have parallel universes :)

Relaxed Memory Order

So whare are relaxed orders good for?

Good if not communications between threads (just do the atomic operation)

- Event counters
- Reference counting

Event counter

```
std::atomic<int> x = 0;
```

```
for (int k = 0; k < 10; ++k)
  x.fetch_add(1, relaxed)
```

```
for (int k = 0; k < 10; ++k)
  x.fetch_add(1, relaxed)
```

```
for (int k = 0; k < 10; ++k)
  x.fetch_add(1, relaxed)
```

What are the possible values of x at the end?

Event counter

```
std::atomic<int> x = 0;
```

```
for (int k = 0; k < 10; ++k)
  x.fetch_add(1, relaxed)
```

```
for (int k = 0; k < 10; ++k)
  x.fetch_add(1, relaxed)
```

```
for (int k = 0; k < 10; ++k)
  x.fetch_add(1, relaxed)
```

What are the possible values of x at the end?

- `assert(x == 30)`

Event counter

```
std::atomic<int> x = 0, y = 0;
```

```
for (int k = 0; k < 10; ++k)
  x.fetch_add(1, relaxed)
  y.fetch_add(1, relaxed)
```

```
for (int k = 0; k < 10; ++k)
  int v0 = x.fetch_add(1, relaxed)
  int v1 = x.load(relaxed)
  printf("%d,", v1);
```

```
for (int k = 0; k < 10; ++k)
  int v0 = x.fetch_add(1, relaxed)
  int v1 = y.load(relaxed)
  printf("%d,", v1);
```

What can you assert?

Event counter

```
std::atomic<int> x = 0, y = 0;
```

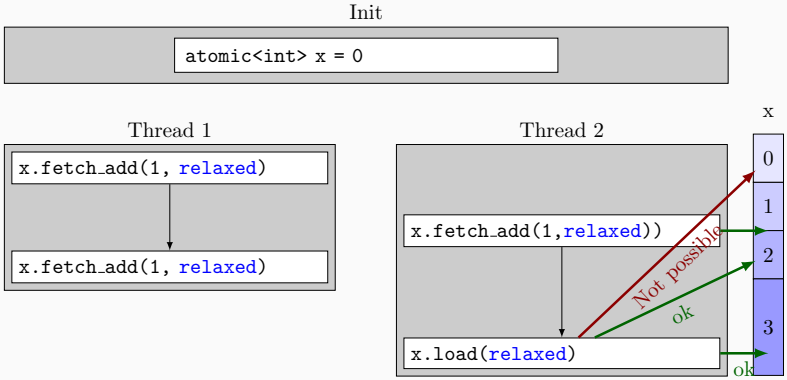
```
for (int k = 0; k < 10; ++k)
  x.fetch_add(1, relaxed)
  y.fetch_add(1, relaxed)
```

```
for (int k = 0; k < 10; ++k)
  int v0 = x.fetch_add(1, relaxed)
  int v1 = x.load(relaxed)
  printf("%d,", v1);
```

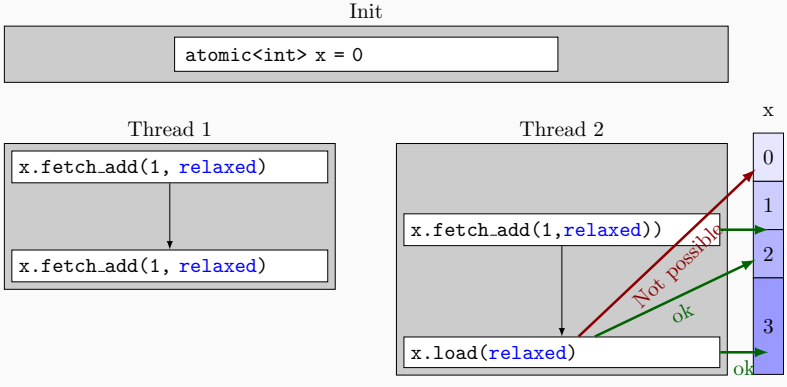
```
for (int k = 0; k < 10; ++k)
  int v0 = x.fetch_add(1, relaxed)
  int v1 = y.load(relaxed)
  printf("%d,", v1);
```

What can you assert?

- `assert(x == 30)`
- `assert(y == 10)`
- In thread 2: `assert(v0+1 <= v1)`
  (RMW operations always see the last stored value)
- In thread 2: `v1` are strictly increasing
  {0, 1, 2, 4, 5, 8 . . . } ~~{0, 1, 1, 4, 5, . . . }~~
- In thread 3: `v1` are weakly increasing
  {0, 0 ,0, . . . , 0} ~~{0, 2, 1 . . . }~~



Even in a parallel universe, the future of the past cannot be before the past



Even in a parallel universe, the future of the past cannot be before the past

Reference counting

```
class shared_vector
{
  void inc_ref()
  {
    FIXME
  }

  void dec_ref()
  {
    FIXME
  }

  std::atomic<int> count;
  std::vector<T>* obj;
};
```

Reference counting

```
void inc_ref()
{
  count++;
}
```

Reference counting

```
void inc_ref()
{
  count++;
}

void dec_ref()
{
  if (--count == 0)
    delete obj;
}
```

Do we need SC strong memory order?

Reference counting

```
void inc_ref()
{
  count.fetch_add(1, relaxed);
}
```

OK…

Reference counting

```
void inc_ref()
{
  count.fetch_add(1, relaxed);
}
```

OK…

```
void dec_ref()
{
  if (count.fetch_sub(1, relaxed) - 1 == 0)
    delete obj;
}
```

FAILS ! Because…

Reference counting

```
void inc_ref()
{
  count.fetch_add(1, relaxed);
}
```

OK…

```
void dec_ref()
{
  if (count.fetch_sub(1, relaxed) - 1 == 0)
    delete obj;
}
```

FAILS ! Because…

Of a data race.
(note: it is not about the counter that reads the right value)

```
auxdata_t* aux;
atomic<int> refcnt = 2;
```

```
aux->use_me();

int c = refcnt.fetch_sub(1, relaxed)
if (c == 1)
  delete aux;
```

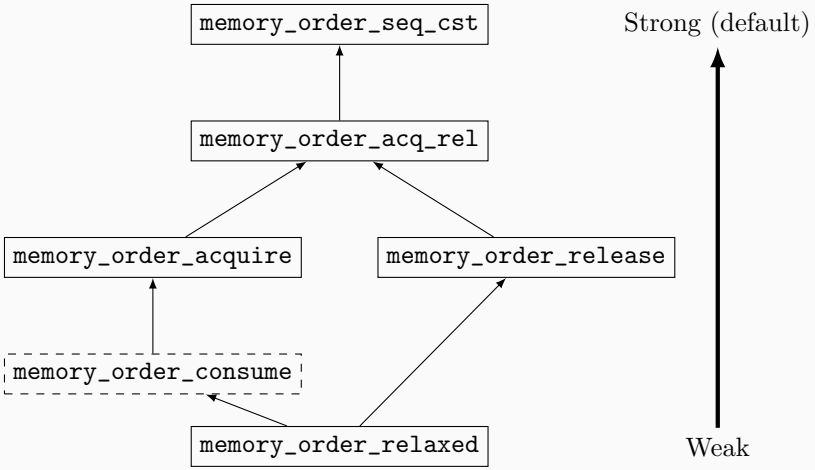
```
aux->use_me();

int c = refcnt.fetch_sub(1, relaxed)
if (c == 1)
  delete aux;
```

No inter-thread synchonisation & no memory ordering constraints:

- one thread can still access object
- another can delete the object

(There is no **happens-before** between `delete aux` and `aux->use_me()` of the two threads)

Non-SC memory order



(Non SC) Acquire-release ordering

Release	store = release on the memory location
Acquire	load = acquire on the memory location
Acquire-Release	Read-Modify-Write = acquire/release on the memory location

(Non SC) Acquire-release ordering

Release	store = release on the memory location
Acquire	load = acquire on the memory location
Acquire-Release	Read-Modify-Write = acquire/release on the memory location

Like for fences:

Acquire operation:

- No read or writes can be redordered before this operation
- Make visible **previous** writes (push)

Release operation:

- No read or write can be reordered after this operation
- See what has been commited (pull)

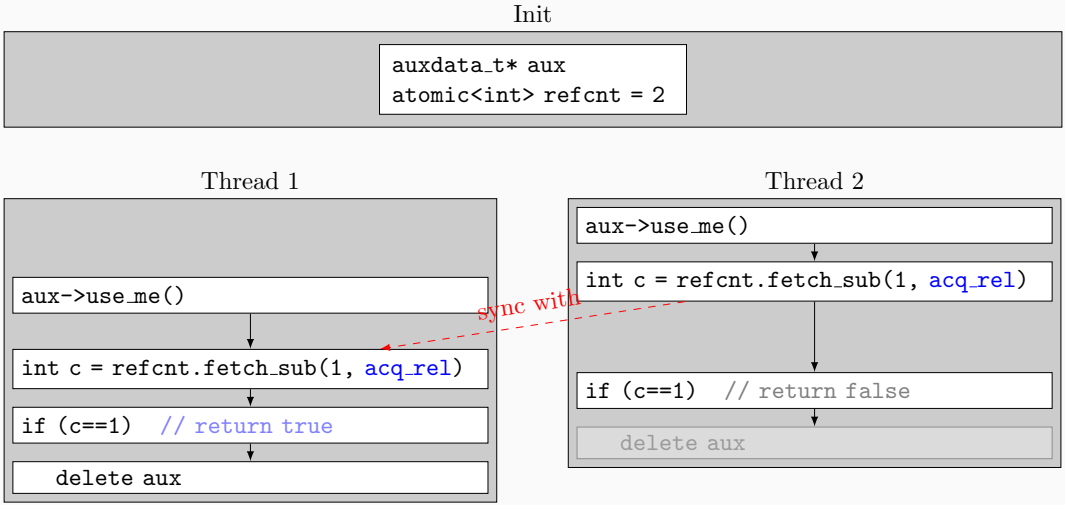
Reference counting

```
auxdata_t* aux;
atomic<int> refcnt = 2;
```

:::

```
aux->use_me();

int c = refcnt.fetch_sub(1, acq_rel)
if (c == 1)
  delete aux;
```

```
aux->use_me();

int c = refcnt.fetch_sub(1, acq_rel)
if (c == 1)
  delete aux;
```


Init
auxdata_t* aux
atomic<int> refcnt = 2
Thread 1
aux->use_me()
int c = refcnt.fetch_sub(1, acq_rel)
if (c==1) // return true
delete aux
sync with
Thread 2
aux->use_me()
int c = refcnt.fetch_sub(1, acq_rel)
if (c==1) // return false
delete aux

Or with fences?

```
aux->use_me();

std::atomic_thread_fence(memory_order_release);
int c = refcnt.fetch_sub(1, memory_order_relaxed);
if (c == 1) {
  std::atomic_thread_fence(memory_order_acquire);
  delete aux;
}
```

(Non-SC) Acquire-release ordering

```
auto relaxed = std::memory_order_relaxed;
int account = 0;
std::atomic<bool> has_been_payed = false;
```

```
account += 2500;
has_been_payed.store(true, release);
```

```
if (has_been_payed.load(acquire))
   assert(account > 0)
```

Data race? Does the assert raise?

(Non-SC) Acquire-release ordering

```
auto relaxed = std::memory_order_relaxed;
int account = 0;
std::atomic<bool> has_been_payed = false;
```

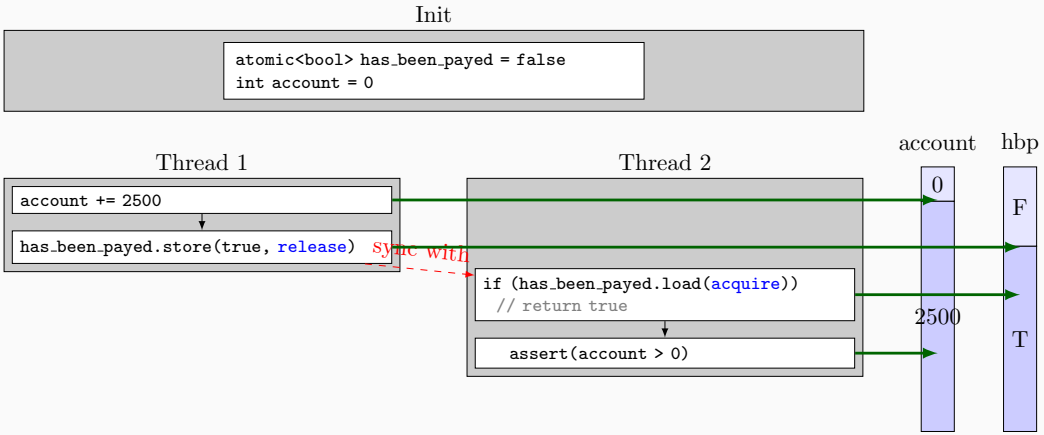
```
account += 2500;
has_been_payed.store(true, release);
```

```
if (has_been_payed.load(acquire))
   assert(account > 0)
```

Data race? Does the assert raise?

- Data race? No
- OK? Yes

Acquire/release on the atomic, syncs and orders R/W accesses to `account`



The two threads now see the same ordering of events

(Non-SC) Acquire-release ordering

```
int account = 0;
std::atomic<bool> has_been_payed = false;
std::atomic<bool> flags = false;
```

```
account += 2500;
has_been_payed.store(true, release);
```

```
while (!has_been_payed.load(acq))
  ;
flag.store(true, release);
```

```
while (!flag.load(acquire))
  ;
assert(account > 0)
```

Data race? Does the assert raise?

(Non-SC) Acquire-release ordering

```
int account = 0;
std::atomic<bool> has_been_payed = false;
std::atomic<bool> flags = false;
```

```
account += 2500;
has_been_payed.store(true, release);
```

```
while (!has_been_payed.load(acq))
  ;
flag.store(true, release);
```

```
while (!flag.load(acquire))
  ;
assert(account > 0)
```

Data race? Does the assert raise?

- Data race? No
- OK? Yes

Transitivity of the “happens-before” relationship !

(Non-SC) Acquire-release ordering

```
int account = 0;
std::atomic<bool> status = 0;
```

```
account += 2500;
status.store(1, release);
```

```
int val;
do {
  val = 1;
} while (status.CAS(val, 2, relaxed))
```

```
while (status.load(acquire) != 2)
  ;
assert(account > 0)
```

Data race? Does the assert raise?

(Non-SC) Acquire-release ordering

```
int account = 0;
std::atomic<bool> status = 0;
```

```
account += 2500;
status.store(1, release);
```

```
int val;
do {
  val = 1;
} while (status.CAS(val, 2, relaxed))
```

```
while (status.load(acquire) != 2)
  ;
assert(account > 0)
```

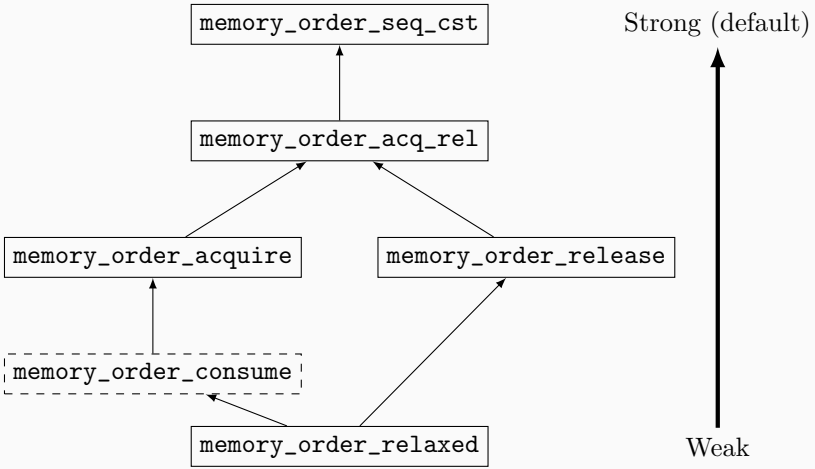
Data race? Does the assert raise?

- Data race? No
- OK? Yes

Release sequence

a load/acquire sync with a previous store/release even if there is a chain of RMW operations (whatever memory ordering)

Sequential Consistency



(Non-SC) Acquire-release looks great. Why one would need full SC ?

SC Example

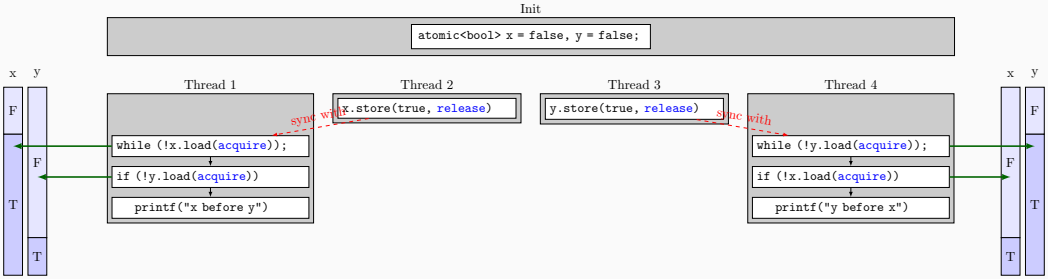
```
std::atomic<bool> x = false, y = false;
```

```
x.store(true, release)
```

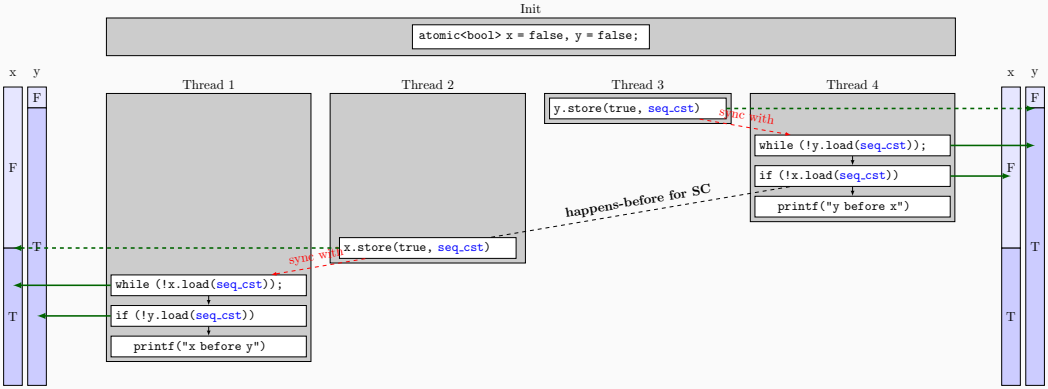
```
y.store(true, release)
```

```
while (!x.load(acquire))
  ;
if (!y.load(acquire))
  printf("x before y");
```

```
while (!y.load(acquire))
  ;
if (!x.load(acquire))
  printf("y before x");
```



- No global sync (acquire/release only syncs the threads doing it)
- We can still have parallel universes



Remind that CS = global sync = total order = up-to-date values = wonderful world

END

Next course: use them for efficient concurrent programming

Questions?