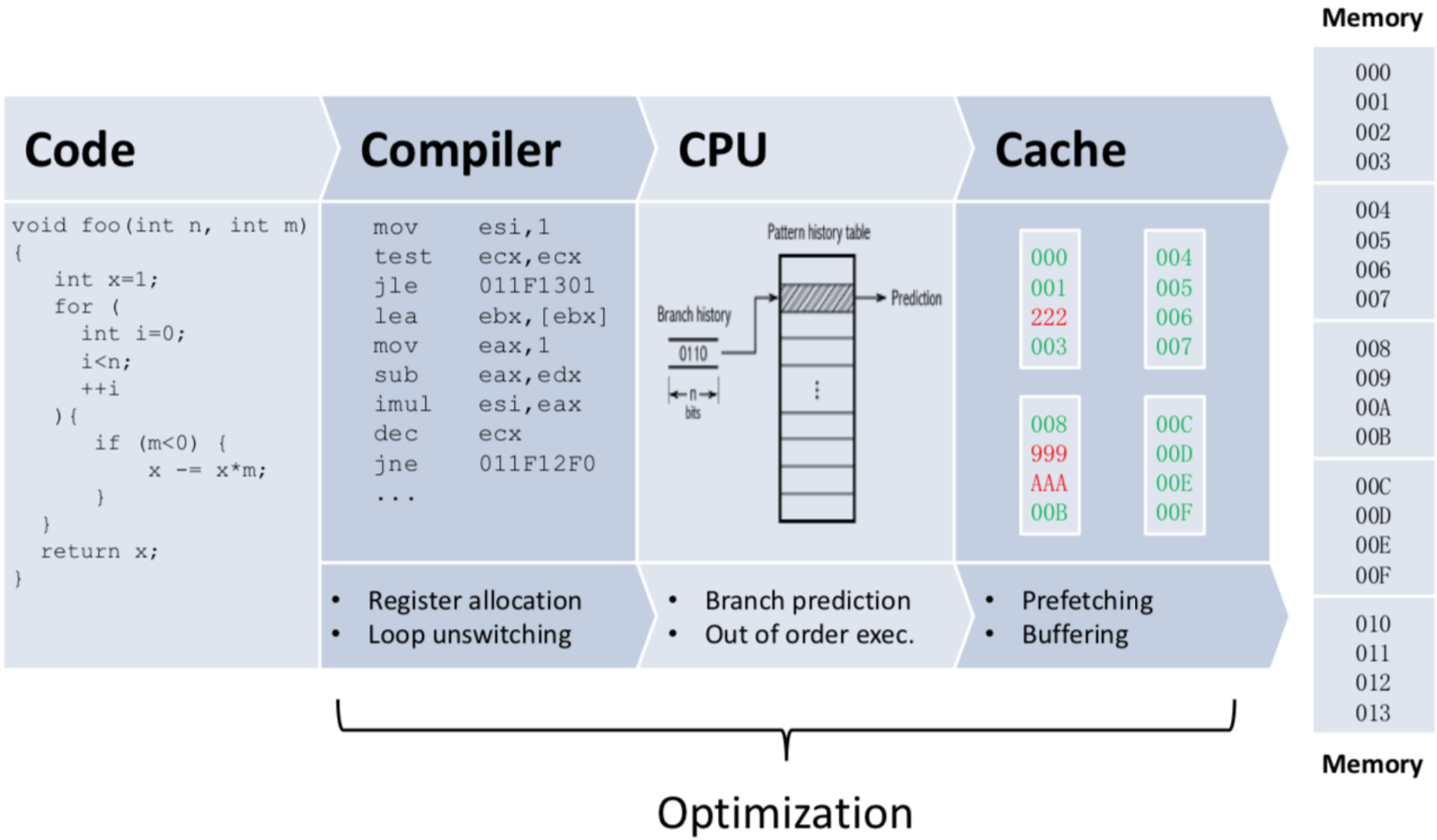
Atomic and memory ordering in modern C++

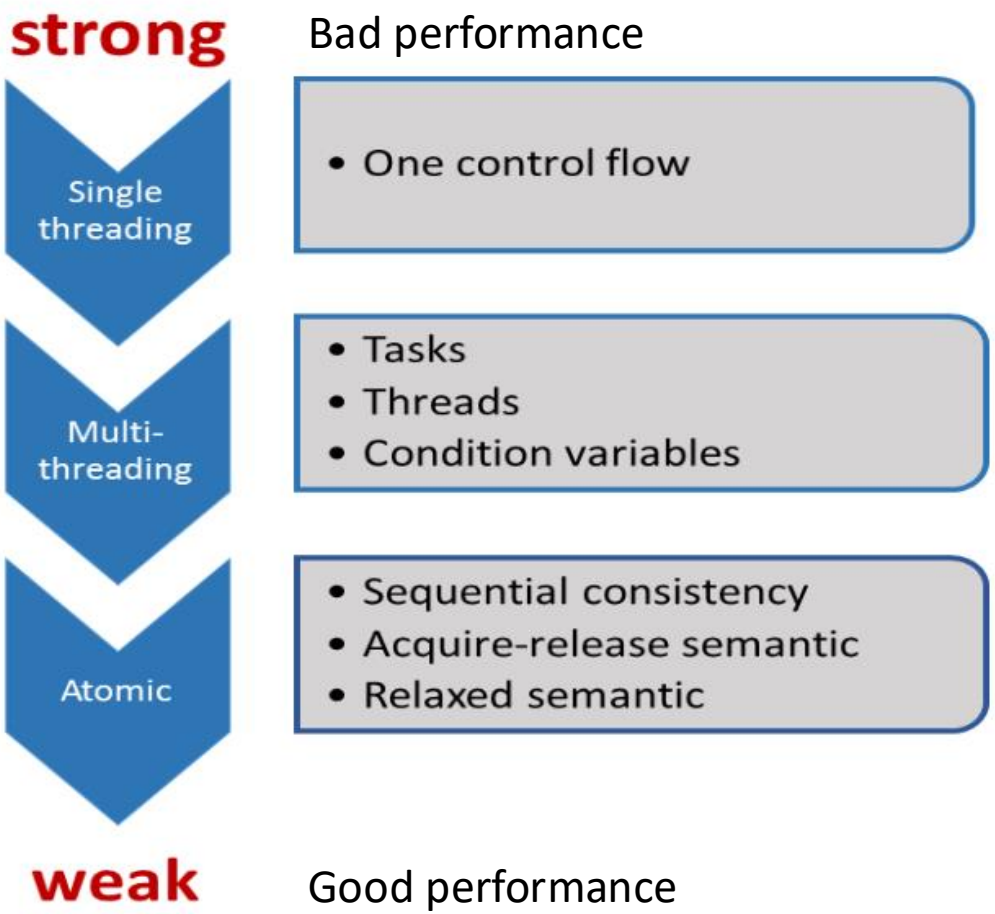
Jui-Hung Hung

How your code is executed?

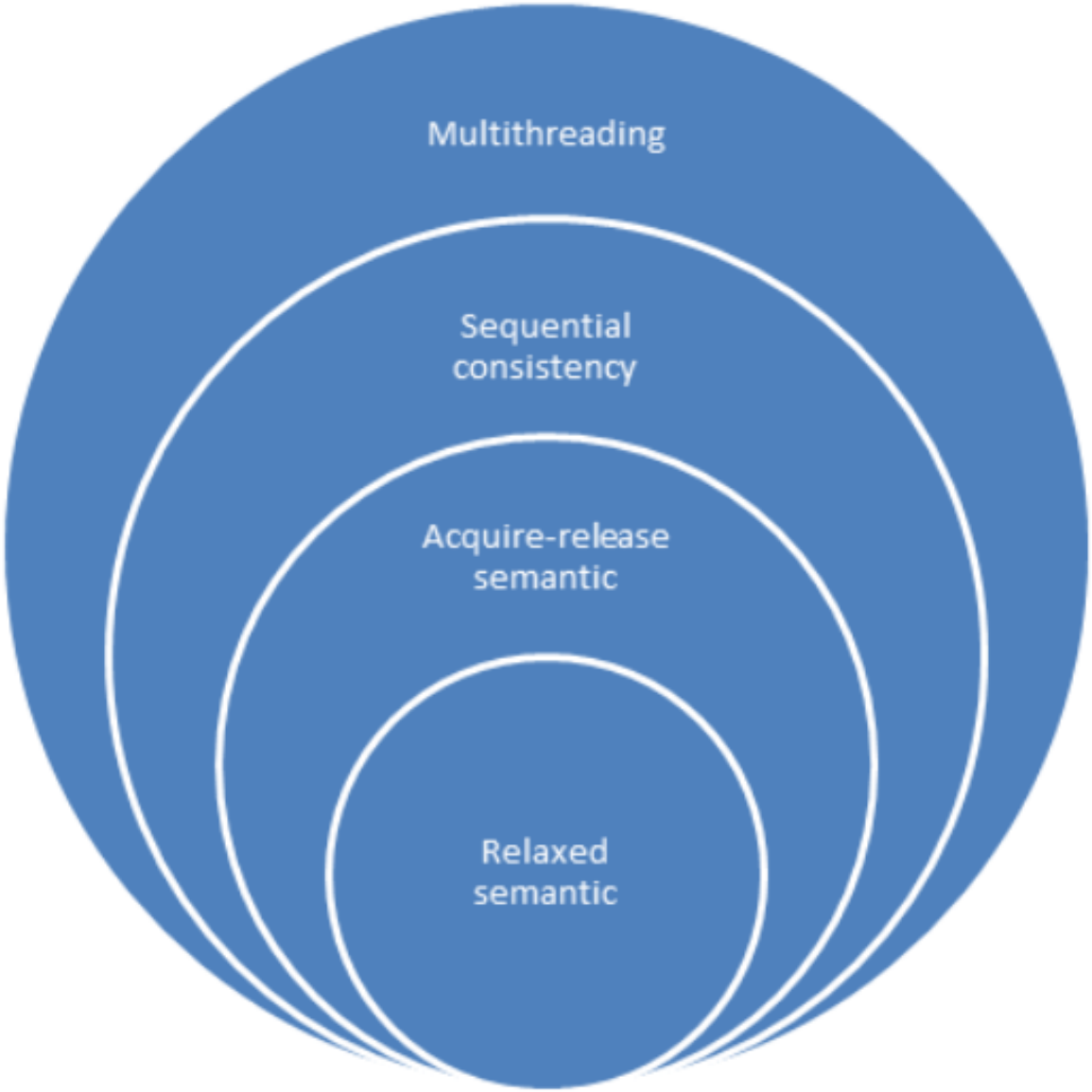


The contract

- C++ memory model defines a contract
- This contract is established between the programmer and the system
- The weaker the rules are the programmer has to follow, the more potential is there for the system to generate a highly optimized executable

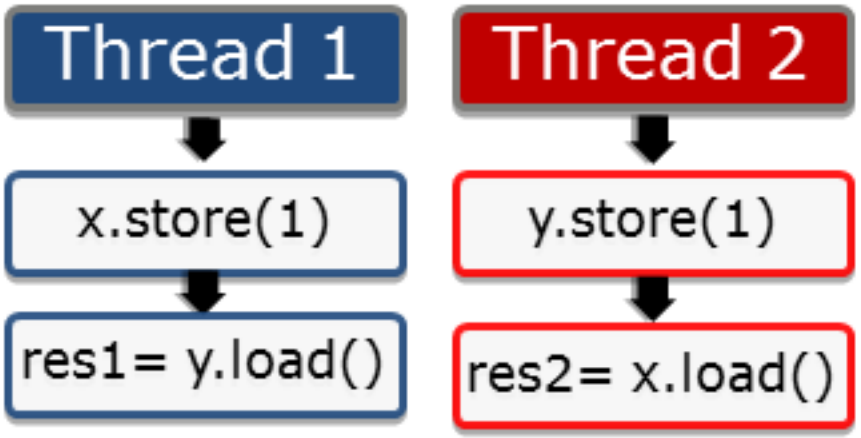


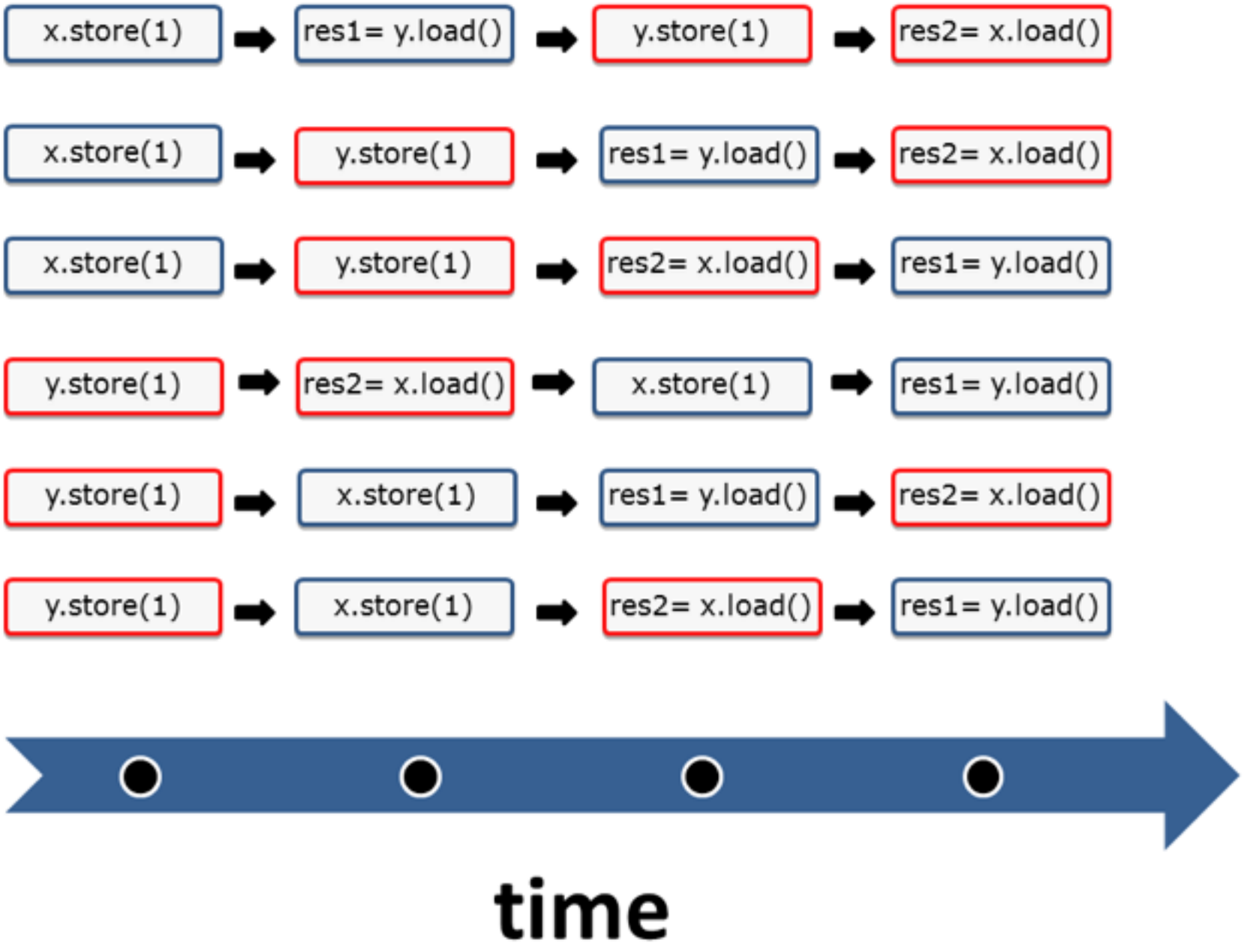
Expert levels



Sequential Consistency

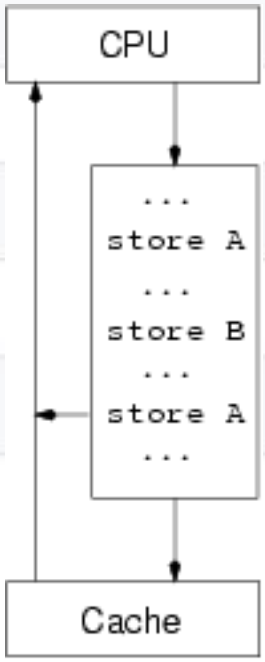
- The result of any execution is the same as-if
  - The instructions of a program are executed in source code order
  - There is a global order of all operations on all threads




x.store(1)
res1= y.load()
y.store(1)
res2= x.load()
x.store(1)
y.store(1)
res1= y.load()
res2= x.load()
x.store(1)
y.store(1)
res2= x.load()
res1= y.load()
y.store(1)
res2= x.load()
x.store(1)
res1= y.load()
y.store(1)
x.store(1)
res1= y.load()
res2= x.load()
y.store(1)
x.store(1)
res2= x.load()
res1= y.load()
time

Hardware memory model

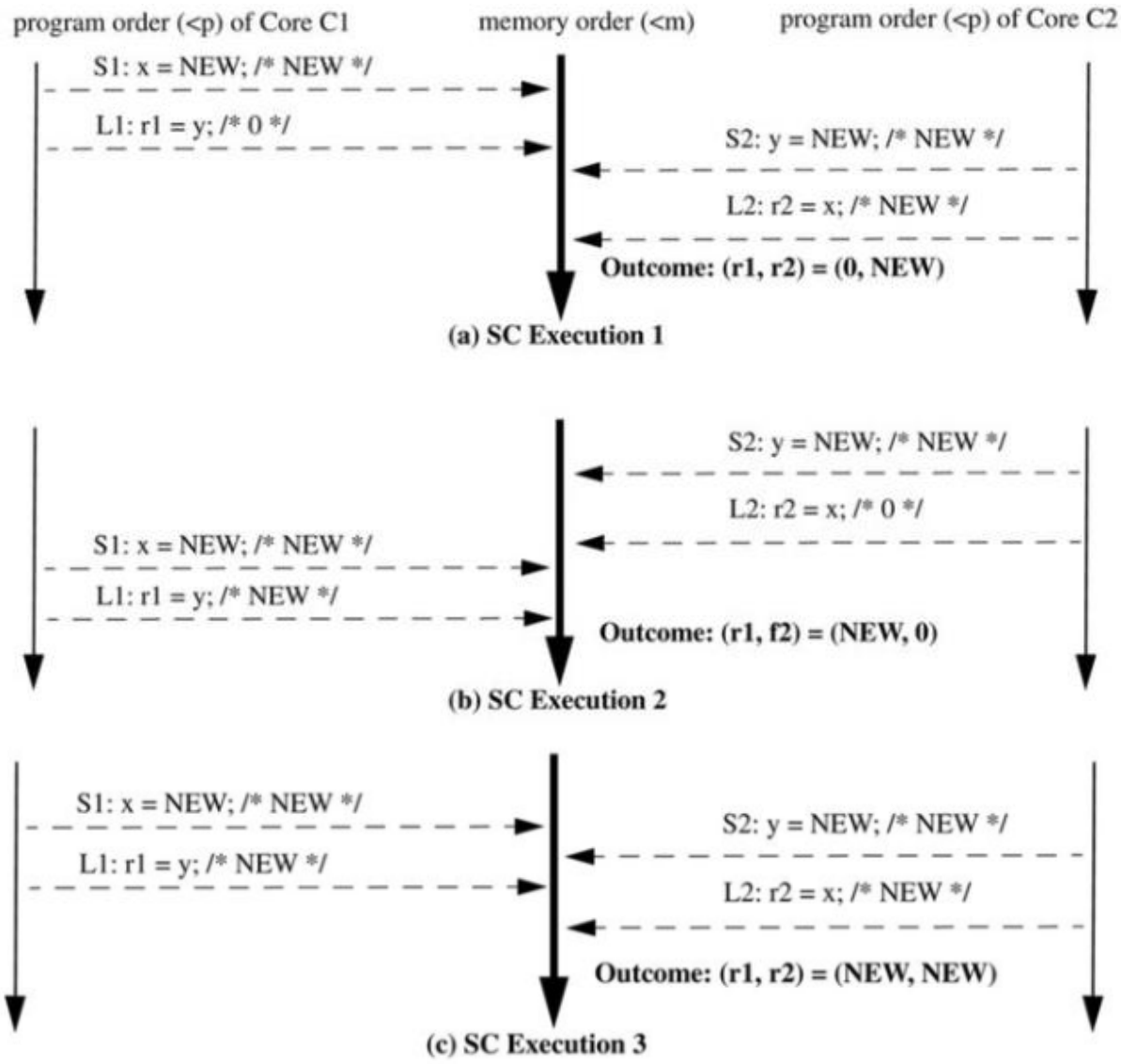
Architecture	Memory Model
x86_64	Total Store Order
Sparc	Total Store Order
ARMv8	Weakly Ordered
PowerPC	Weakly Ordered
MIPS	Weakly Ordered



Memory barrier

SC

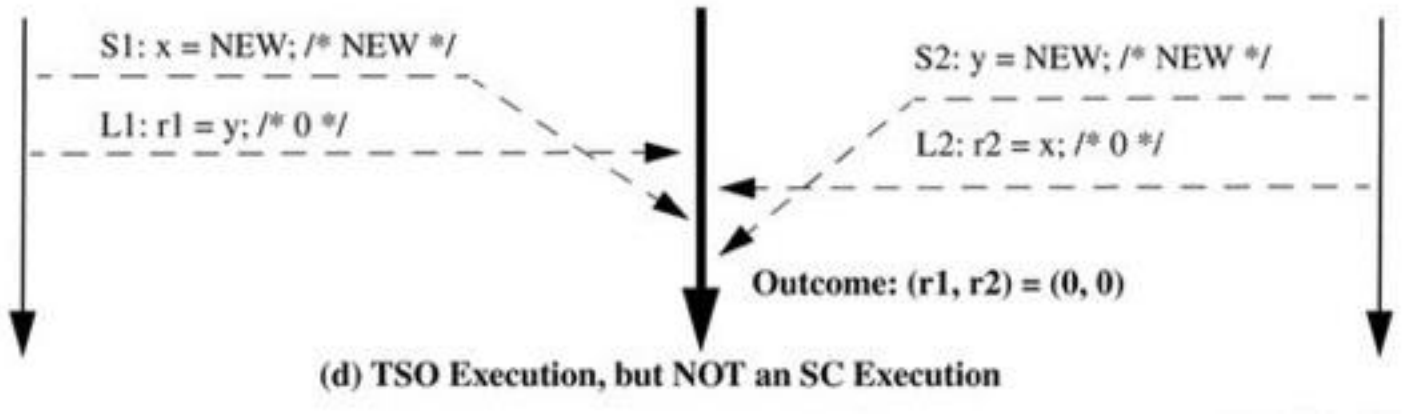
TABLE 3.3: Can Both r1 and r2 be Set to 0?		
Core C1	**Core C2**	**Comments**
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	/* Initially, x = 0 & y = 0*/



TSO

TABLE 3.3: Can Both r1 and r2 be Set to 0?		
Core C1	**Core C2**	**Comments**
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	/* Initially, x = 0 & y = 0*/

- Acquire-release semantic



(d) TSO Execution, but NOT an SC Execution

Your load get value from the closest same-core store, not that from the global store

Weak memory model

- Relaxed semantic
- Free for optimization
- No picture of a global order

C++ guarantees you

- **Atomic operations**
  - Operations, which will be executed without interruption
- **The partial order of operations**
  - Sequence of operations, which can not be changed
- **Visible effects of operations**
  - Guarantees, when an operation on shared variables will be visible in another thread
- C++ code now has a standardized library to call regardless of who made the compiler and on what platform it's running.
- There's a standard way to control how different threads talk to the processor's memory

- Memory model is still evolving!

C++ memory order

```
namespace std {
 typedef enum memory_order {
   memory_order_relaxed,
   memory_order_consume,
   memory_order_acquire,
   memory_order_release,
   memory_order_acq_rel,
   memory_order_seq_cst
 } memory_order;
}
```

- **memory_order_seq_cst**: guarantees SC; default
- **memory_order_acquire**: guarantees that subsequent loads are not moved before the current load or any preceding loads.
- **memory_order_release**: preceding stores are not moved past the current store or any subsequent stores.
- **memory_order_acq_rel**: combines the two previous guarantees.
- **memory_order_consume**: potentially weaker form of memory_order_acquire that enforces ordering of the current load before other operations that are data-dependent on it
- **memory_order_relaxed**: all reorderings are okay.

Atomic<>

- You need atomic type values coupled with a memory model to achieve lock-free programming
- Compiler guarantees atomicity
  - unlike volatile
- Atomic Operations
  - Store/load/exchange/compare_exchange
  - fetch_add / fetch_sub / fetch_or / fetch_xor / fetch_and
  - You can explicit give memory order parameter
    - Use functions with the “_explicit” suffix

Dekker's algorithm

```
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn    = 0   // or 1
```

```
P0:
   flag[0] = true;
   while (flag[1] == true) {
      if (turn ≠ 0) {
         flag[0] = false;
         while (turn ≠ 0) {
           // busy wait
         }
         flag[0] = true;
      }
   }

   // critical section
   ...
   turn    = 1;
   flag[0] = false;
   // remainder section
```

```
P1:
   flag[1] = true;
   while (flag[0] == true) {
      if (turn ≠ 1) {
         flag[1] = false;
         while (turn ≠ 1) {
           // busy wait
         }
         flag[1] = true;
      }
   }

   // critical section
   ...
   turn    = 0;
   flag[1] = false;
   // remainder section
```

Dekker's algorithm with Atomic

```
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
}
```

Caveats

- As soon as atomic operations that are not tagged memory_order_seq_cst enter the picture, the sequential consistency guarantee for the program is lost
- Memory order (like fences) controls other non-atomic operations not the atomic operation itself
- In many cases, memory_order_seq_cst atomic operations are reorderable with respect to other atomic operations performed by the same thread
- Total sequential ordering requires a full memory fence CPU instruction on all multi-core systems. This may become a performance bottleneck since it forces the affected memory accesses to propagate to every core

Acquire-release semantic

- Happens-before relation (v.s. sequenced-before)
  - If SA release-stores a value v in A and LA acquire-loads that value, then SA happens before LA.
- If an atomic store in thread A is tagged memory_order_release and an atomic load in thread B from the same variable is tagged memory_order_acquire, **ALL** memory writes (non-atomic and relaxed atomic) that *happened-before* the atomic store from the point of view of thread A, become *visible side-effects* in thread B.
  - That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory.
- Partial Ordering
  - Synchronization per variable
  - Concurrent operations are possible
- The synchronization is established only between the threads *releasing* and *acquiring* the same atomic variable.

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

std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p  = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))
        ;
    assert(*p2 == "Hello"); // never fires
    assert(data == 42); // never fires
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

Release-Consume ordering

- If an atomic store in thread A is tagged memory_order_release and an atomic load in thread B from the same variable that read the stored value is tagged memory_order_consume, **ALL** memory writes (non-atomic and relaxed atomic) that *happened-before* the atomic store from the point of view of thread A, become *visible side-effects* within those operations in thread B into which the load operation *carries dependency*
- that is, once the atomic load is completed, those operators and functions in thread B that use the value obtained from the load are guaranteed to see what thread A wrote to memory.

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

std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p  = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_consume)))
        ;
    assert(*p2 == "Hello"); // never fires: *p2 carries dependency from ptr
    assert(data == 42); // may or may not fire: data does not carry dependency from ptr
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

ABA problem

- In multithreaded computing, the **ABA problem** occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed".
- However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption.
- A common case of the ABA problem is encountered when implementing a lock-free data structure.

Don't program lock-free

- **Herb Sutter**: Lock-free programming is like playing with knives.
- **Anthony Williams**: "*Lock-free programming is about how to shoot yourself in the foot.*"
- **Tony Van Eerd**: "*Lock-free coding is the last thing you want to do*."
- **Fedor Pikus**: "*Writing correct lock-free programs is even harder.*"
- **Harald Böhm**: "*The rules are not obvious.*"

Let’s wait few more years…