C++ Memory Model Tutorial

Wenzhu Man

Outline

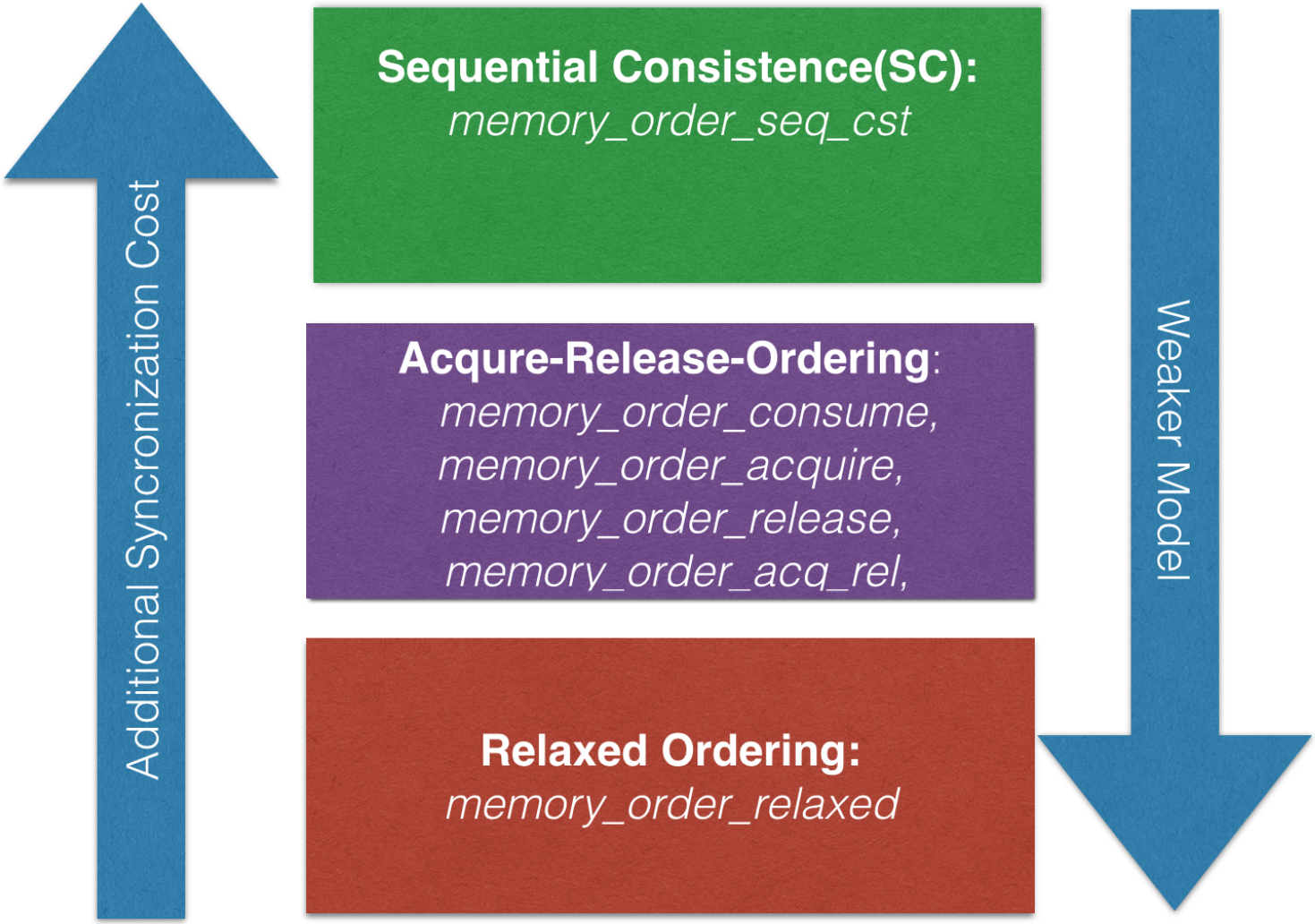
1. **Motivation**

2. **Memory Ordering for Atomic Operations**
   - The synchronizes-with and happens-before relationship (not from lecture / extra material)
   - Sequential Consistent Ordering(Default)
   - Relaxed ordering
   - Acquire-release Ordering

Motivation

- Portability
  - Threading with Pthread works but: relies on the compiler. No guaranteed portability.
  - Same code - different compiler - different compiler optimizations different hardware - Same behavior
- Performance
  - Simple usage of atomic instructions
  - No locking for atomic execution of e.g. increment

Different Ordering Options C++ Memory Model offers



The synchronizes-with and happens-before relationship(not from lecture)

Synchronizes-with Relationship

- A suitably tagged atomic write operation W on a variable x **synchronizes with** a suitably tagged atomic read operation on x that reads the value stored by either that write (W), or a subsequent atomic write operation on x by the same thread that performed the initial write W, or a sequence of atomic read-modify- write operations on x by **any thread**, where the value read by the first thread in the sequence is the value written by W[1]
- suitably tagged eg.memory_order_seq_cst(default)

[1] **"C++ Concurrency In Action", PRACTICAL MULTITHREADING, ANTHONY WILLIAMS.**

The synchronizes-with and happens-before relationship(not from lecture)

Happens-before Relationship

- Let A and B represent operations performed by a multithreaded process. If A happens-before B, then the memory effects of A effectively become visible to the thread performing B before B is performed.[2]
- Intra-thread happens-before equals sequenced before
- Inter-thread happens-before relies on the synchronizes-with relationship
- Happens-before relationship is transitive

[2] **http://preshing.com/20130702/the-happens-before-relation/.**

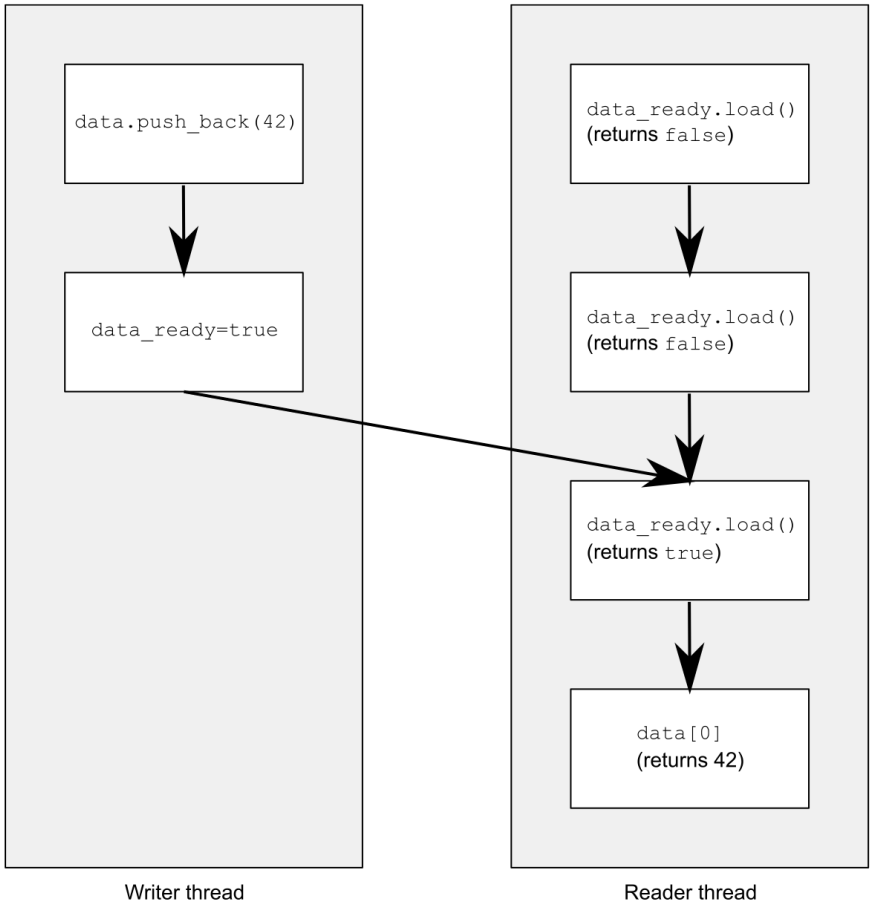
The synchronizes-with and happens-before relationship

```
#include <vector>
#include <atomic>
#include <iostream>
std::vector<int> data;
std::atomic<bool> data_ready(false);
void reader_thread()
{
    while(!data_ready.load())       //1
    {
        std::this_thread::sleep(std::milliseconds(1));
    }
    std::cout<<"The answer="<<data[0]<<"\n"; //2
}

void writer_thread()
{
    data.push_back(42); //3
    data_ready=true;    //4
}
```

- Sequenced before(intra-thread):1 happens before 2 , 3 happens before 4
- When the value read from data_ready is true, the write synchronizes-with that read, creating a happens-before relationship :4 happens before 1

The synchronizes-with and happens-before relationship



3

[3] **"C++ Concurrency In Action", PRACTICAL MULTITHREADING, ANTHONY WILLIAMS.**

Sequential Consistent Ordering(Default)

```cpp
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
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;
}
```

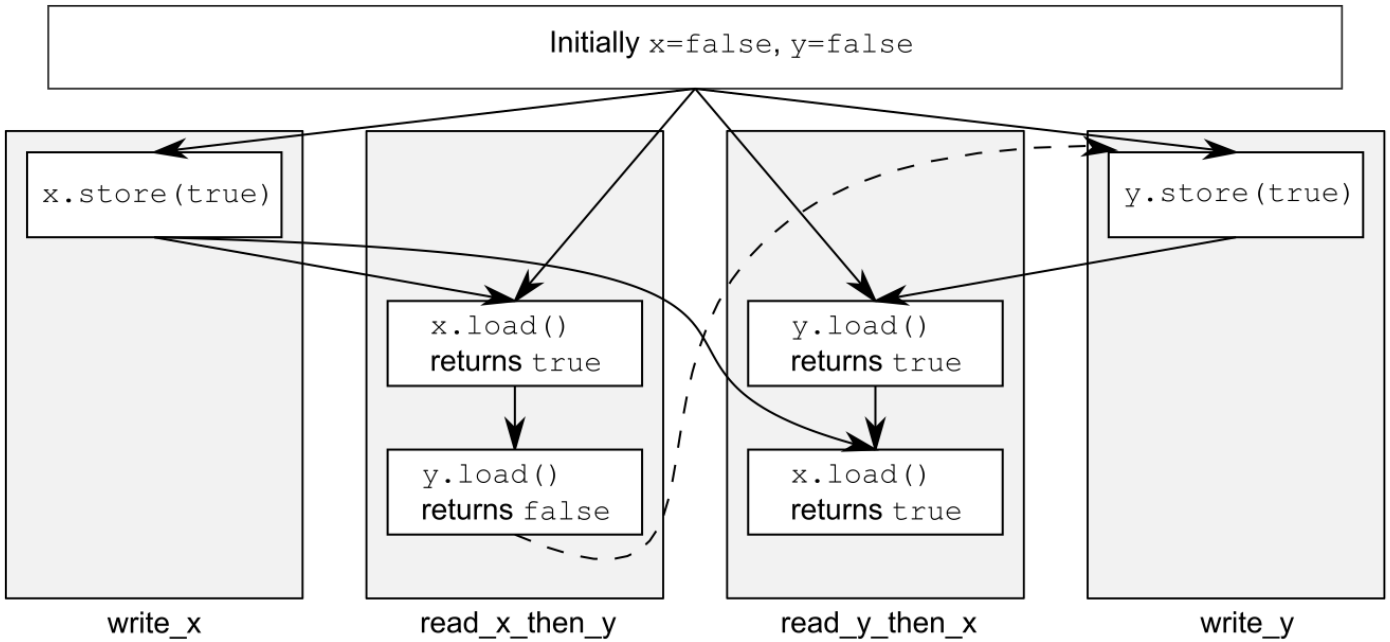
```cpp
int main() {
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);
}
```

What is the output?

Sequential Consistent Ordering(Default)

answer: the assertion will never fail(z could be 1 or 2).

One possible Order and Happens-before relation



4

[4]"C++ Concurrency In Action", **PRACTICAL MULTITHREADING, ANTHONY WILLIAMS**.

Relaxed ordering

- Operations can be freely reordered provided they obey any happens-before relationships they are bound
- Do not introduce synchronizes-with relationships
- if we change the operations in former example from memory_order_seq_cst to memory_order_relaxed, what's the possible output?

Acquire-release Ordering

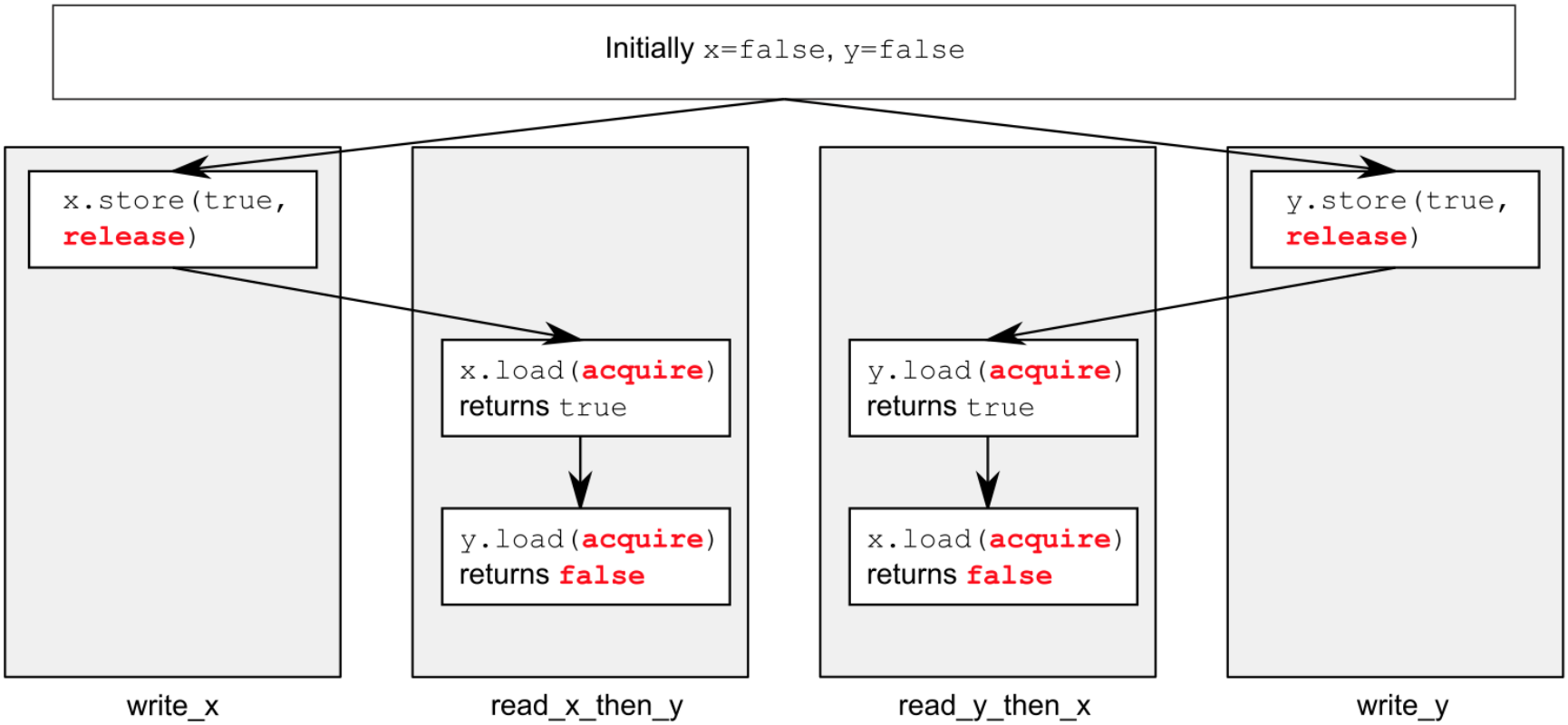
- A release operation synchronizes-with an acquire operation that reads the value written.

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x()
{
    x.store(true,std::memory_order_release);
}
void write_y()
{
    y.store(true,std::memory_order_release);
}
void read_x_then_y()
{
    while(!x.load(std::memory_order_acquire));
    if(y.load(std::memory_order_acquire))
    ++z;
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));
    if(x.load(std::memory_order_acquire))
        ++z;
}
```

```
int main() {
    x=false;
    y=false;
    z=0;
    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);
}
```

Acquire-release Ordering

One possible Output and Happens-before Relation



5

[5] **"C++ Concurrency In Action", PRACTICAL MULTITHREADING, ANTHONY WILLIAMS.**

Acquire-release Ordering

- Acquire-release operations can impose ordering on relaxed operations
- what is the possible output?

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);
    y.store(true,std::memory_order_release);
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));
    if(x.load(std::memory_order_relaxed))
        ++z;
}
int main() {
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

Ordering nonatomic operations with atomics

What is the output?

```
#include <atomic>
#include <thread>
#include <assert.h>
//x is now a plain nonatomic variable
bool x=false;
std::atomic<bool> y;
std::atomic<int> z;
void write_x_then_y(){
    //1:Store to x before the fence
    x=true;
    std::atomic_thread_fence(std::memory_order_release);
    //2:Store to y after the fence
    y.store(true,std::memory_order_relaxed);
}
void read_y_then_x(){
    //Wait until you see the write from #2
    while(!y.load(std::memory_order_relaxed));
    std::atomic_thread_fence(std::memory_order_acquire);
    //This will read the e value written by #1
    if(x)
      ++z;
}
int main() {
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

Question

Dekker and Peterson's Algorithm

Thread 1:

```
flag1 = 1;          // a: declare intent
if( flag2 != 0 )    // b
   // resolve contention
else
   // enter critical section
```

Thread 2:

```
flag2 = 1;          // c: declare intent
if( flag1 != 0 )    // d
   // resolve contention
else
   // enter critical section
```

- Could both threads enter the critical region(flags are shared and a plain nonatomic variable, initially zero)?
- If so, solution(under C++ Memory Model)?