Memory consistency in C++

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Francisco Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

C++ and memory consistency

- C++11 defines its own **concurrency model** as part of the language.
- **Goal**: Avoid the need to write code in lower level languages (C, assembler, . . .) to obtain better performance.
  - Atomic types.
  - Low level synchronization mechanisms.
- Allows to build **lock free data structures**.

Objects and memory locations

- **Object**: Is a storage region.
  - A sequence of one or more bytes.

- **Memory location**: Is an object of scalar type or a sequence of contiguous bit fields.

- **An object is stored in one or more memory locations**.

Example

- Structure:

```
struct {
  int i;
  char c;
  int d: 10;
  int e: 16;
  double f;
};
```

- Memory locations:
  1. **i**.
  2. **c**.
  3. **d**, **e**.
  4. **f**.

Rules

- Two threads may access to **different memory locations** simultaneously.
- Two threads may access to the **same memory locations** simultaneously if both accesses are for **reading**.
- If two threads try to access simultaneously to the **same memory location** and any access is a **write**, there is a **potential race condition**.
  - Depends on whether an **ordering between both accesses** is stablished.

Ordering and race conditions

- **Classic solution**: Use **synchronization** mechanisms.
  - Allow to guarantee **mutual exclusion**.
  - Based on OS $\rightarrow$ Might be costly.

- **Alternative**: Use **atomic operations** to ensure **ordering**.

  - If **ordering between two accesses** to a memory location is not established,
  - some of the accesses **is not atomic**,
  - and at least one of the accesses is a **write**,
  - those are a **data race** and **program behavior is not defined**.

Modification order

- **Modification order**: Sequence of writes on an object.
  - If two threads see different modification orders on an object there is a **data race**.
  - Modifications do not need to be visible in the same instant in all threads.

- A subsequent read to a write on the same thread observes the written value or a subsequent value in its **modification order**.

Atomic operations

- They are **indivisible operations**.
  - If a thread performs an **atomic read** from a variable and other thread performs an **atomic write** on the same variable and there is no **more threads accessing**:
    - The read returns the **previous value** to the write or the **written value**.
  - If any of the operations (read or write) is **non atomic** the **behavior is not defined**.
    - A value can be obtained that is not the previous or the subsequent one.

Atomic types

- A generic type **atomic<T>** allows to define atomic variables for type **T**, where **T** is:
  - An integral type.
  - A pointer type.
  - Type **bool**.
  - It is undefined for real number types (**float**, **double**).
  - Also available for user defined types fulfilling some constraints.
- All atomic types have a member **is_lock_free()**.
  - Determine if their implementation is **lock-free**.
- Additionally there is a type **atomic_flag**:
  - The only type that is guaranteed to be **lock-free**.

Operations on atomic types

- Operations on atomics may optionally specify a memory order.
  - By default **memory_order_seq_cst**.
- Store operations:
  - **memory_order_relaxed**, **memory_order_release**, **memory_order_seq_cst**.
- Read operations:
  - **memory_order_relaxed**, **memory_order_consume**, **memory_order_acquire**, **memory_order_seq_cst**
- Read-modify-write operations:
  - **memory_order_relaxed**, **memory_order_consume**, **memory_order_acquire**, **memory_order_release**, **memory_order_acq_rel**, **memory_order_seq_cst**.

atomic_flag

- **Most simple possible** atomic type.
  - **Two possible states**: **enabled** o **disabled**.
  - It is always lock-free.
  - Always must be explicitly initiated to disabled.

    ```
    std::atomic_flag f1 = ATOMIC_FLAG_INIT;
    ```

  - **Operations**:
    - **Disable**:

      ```
      f1.clear();
      ```

    - **Enable and check** previous value:

      ```
      f1.test_and_set();
      ```

  - May provide memory order for operation.

Example: A *spin lock*

- Lock not using OS services.
  - Useful for very short lockings when you desire to avoid context switching problems.

spin lock mutex

```
class spinlock_mutex {
private:
  std::atomic_flag f;
public:
  spinlock_mutex() : f{ATOMIC_FLAG_INIT} {}

  void lock() {
    while (f.test_and_set()) {}
  }
  void unlock() {
    flag.clear();
  }
};
```

atomic_bool

- More operations than **atomic_flag**.
- Can be initiated and assigned with **bool**s.
- Cannot be copied from another **atomic<bool>**.
- Modification: **a.store(order)**
- Query: **a.exchange(b, order)**
- Automatic conversion to **bool** (seq. consistency): **a.load(order)**.

Example

```
std::atomic<bool> a;
bool x = a.load(std::memory_order_acquire);
a.store(true);
x = a.exchange(false, std::memory_order_acq_rel);
```

Compare and exchange

- Compares atomic value with an **expected** value.
  - If both are equal, the **desired** value is stored in the atomic.
  - If not equal, atomic is left unmodified.
  - It always returns success/failure indication.

- Two versions:
  1. **a.compare_exchange_weak(e,d)**:
     - Allows spurious failures (context switch) in some architectures.
     - May behave as if ***this!=e** even if they are equal.
  2. **a.compare_exchange_strong(e,d)**:
     - Does not allow for spurious failures.

atomic_address

- Atomic access to a memory address.
- Cannot be copied.
- Can copy a (**void***) pointer.
- Interface similar to **atomic<bool>**:
  - **is_lock_free()**, **load()**, **store()**, **exchange()**, **compare_exchange_weak()**, **compare_exchange_strong()**.
- Additional operations.
  - **fetch_add()**, **fetch_sub()**.
    - Allow for memory ordering specification.
    - Return value previous to change.
- **+=**, **-=**.
  - Return the value after the change.
  - All operations allow byte arithmetic.
- Other arithmetics with **atomic<T*>**.

atomic<integral>

- Can be applied to all integral types.
- General operations:
  - **is_lock_free()**, **load()**, **store()**, **exchange()**, **compare_exchange_weak()**, **compare_exchange_strong()**.
- Arithmetic operations.
  - **fetch_add()**, **fetch_sub()**, **fetch_and()**, **fetch_or()**, **fetch_xor()**.
  - **+=**, **-=**, **&=**, **|=**, **ˆ=**.
  - **++x**, **x++**, **–x**, **x–**
  - There are no other arithmetic operations (*****, /, **%**).

synchronizes-with relation

- **Relationship** between operations on **atomic types**.
- A **write** on an atomic value **synchronizes-with** a **read** on that atomic value **reading that value**:

  i Stored by **that write**.

  ii Stored by a **subsequent write** from the same thread that performed the write.

  iii Stored by a **sequence** of **read-modify-write** operations on the value from any thread in which the first operation read the value stored by the write.

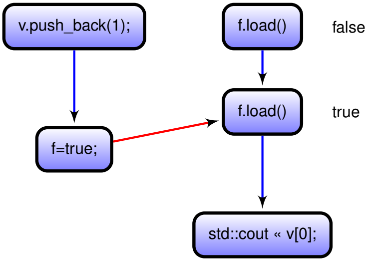
happens-before relationship

- Specified which operations **see the effects** from other operations.
- Within a thread, an operation **happens-before** other operation if it appears in a **preceding sentence**.
  - **There is no order between two operations from the same sentence**.
- Among two threads, an operation in one thread **happens-before** other operation from other thread if:
  i There is a **synchronizes-with** relationship among both operations.
  ii There is a **happens-before** a **synchronizes-with** chain of relationships among both operations.

Ordering: Sequential consistency

Example

```
std::vector<int> v;
std::atomic_bool f(false);

void writer() {
  v.push_back(1); // #1
  f = true;  // #2
}

void reader() {
  while(!f.load()) { // #3
    std::this_thread::sleep(
      std::milliseconds(1));
  }
  std::cout << v[0] << std::endl; // #4
}
```



- Only possible result: **v[0] == 1**.

Sequential consistency

- **memory_order_seq_cst**.
- The program is consistent with a **sequential view**.
- If all the operations on atomics are **sequentially consistent**, multi-threaded program behavior is as if all the operations would be performed in some particular order in a single thread.
- There cannot be reorderings.
- It is the simplest model to reason about.
- It is the most costly model in terms of performance.

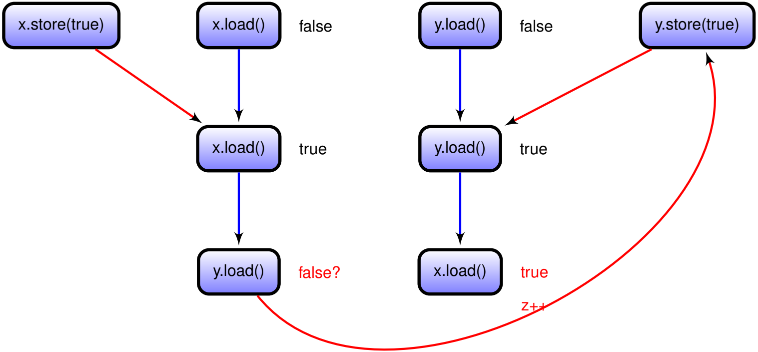
Access

```
std::atomic<bool> x, y;
std::atomic<int> z;

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

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

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

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

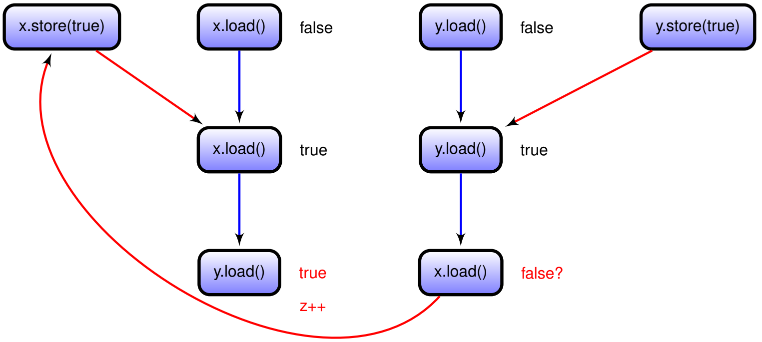
Threads launching

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

  std::thread t1{f};
  std::thread t2{g};
  std::thread t3{h};
  std::thread t4{i};

  t1.join();
  t2.join();
  t3.join();
  t4.join();

  assert(z.load() !=0);

  return 0;
}
```

Sequential consistency: Analysis



Sequential consistency: Analysis



Non-sequentially consistent orders

- There is no **global order of events**.
  - Each thread may have **a different view**.
    - Threads might not agree on the same order of events.
  - But, ...
    - **All threads must agree in the modifications order for each variable**.

- **Alternatives**:
  - **relaxed** ordering.
  - **release/acquire** ordering.

Relaxed ordering

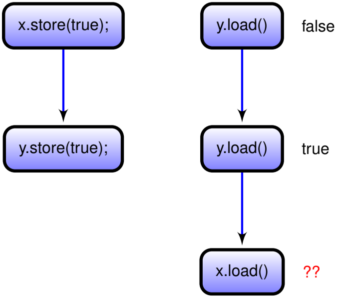
- **memory_order_relaxed**
- Relaxed operations on atomics **do not participate** in **synchronizes-with** relationship.

- Operations on same variable in the same thread **do fulfill** **happens-before** relationship.
  - Accesses to an atomic variable within the same thread **cannot be reordered**.
  - Once a thread has seen a value from variable **it cannot see** an older value of that variable.

Example

Data access

```
std::atomic<bool> x, y; std::atomic<int> z;

void f() {
  x.store(true, std::memory_order_relaxed);
  y.store(true, std::memory_order_relaxed);
}
void g() {
  while (!y.load(std::memory_order_relaxed)) {}
  if (x.load(std::memory_order_relaxed)) { ++z; }
}

int main() {
  x=false; y=false; z=0;
  std::thread t1{f}; std::thread t2{g};
  t1.join(); t2.join();
  return 0;
}
```



Release/acquire ordering

- **memory_order_acquire**, **memory_order_release**, **memory_order_acq_rel**.
- **Intermediate** level of synchronization.
- A **release** operation **writing a value synchronizes-with** an **acquire** operation **reading that value**.
- **Impact**:
  - **Different threads may see different orders**.
  - **Not all orders are possible**.

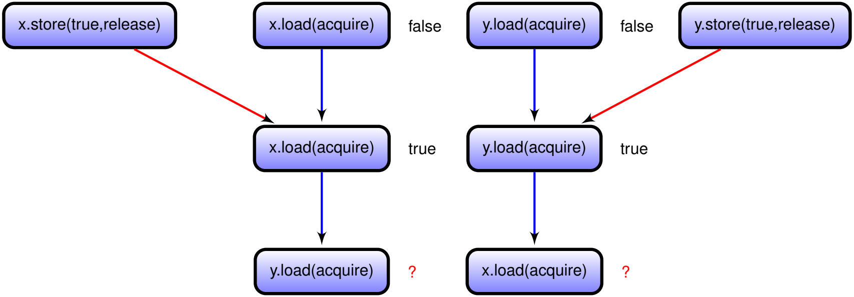
Access

```
std::atomic<bool> x, y;
std::atomic<int> z;

void f() {
  x.store(true, std::memory_order_release);
}

void g() {
  y.store(true, std::memory_order_release);
}

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

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

Threads launching

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

  std::thread t1{f};
  std::thread t2{g};
  std::thread t3{h};
  std::thread t4{i};

  t1.join();
  t2.join();
  t3.join();
  t4.join();

  assert(z.load() !=0);
  return 0;
}
```

Analysis



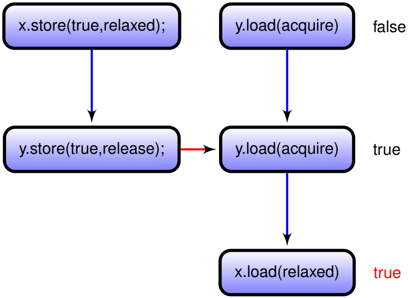
- multiple orders are possible as there is no relationship *acquire* $\rightarrow$ *release* .

Combining orderings

- An **equivalent** effect to **sequential consistency** can be obtained with **lower cost**.

Access

```
std::atomic<bool> x, y; std::atomic<int> z;

void f() {
  x.store(true, std::memory_order_relaxed);
  y.store(true, std::memory_order_release);
}
void g() {
  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 t1{f}; std::thread t2{g};
  t1.join(); t2.join();
  assert(z.load() !=0);

  return 0;
}
```



Barriers

- **Force ordering** without modifying data.

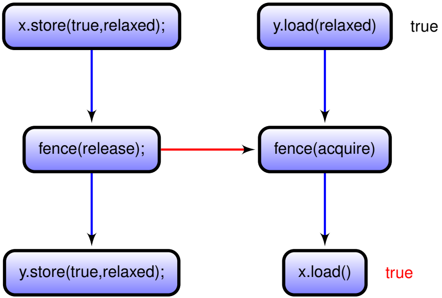
Example

```
std::atomic<bool> x, y;
std::atomic<int> z;

void f() {
  x.store(true, std::memory_order_relaxed);
  std::atomic_thread_fence(std::memory_order_release);
  y.store(true, std::memory_order_relaxed);
}

void g() {
  while (!y.load(std::memory_order_relaxed)) {}
  std::atomic_thread_fence(std::memory_order_acquire);
  if (x.load(std::memory_order_relaxed)) ++z;
}
```

Threads

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

  std::thread t1(f);
  std::thread t2(g);

  t1.join();
  t2.join();

  assert(z.load() !=0);
  return 0;
}
```

Barriers: Analysis



Summary

- The C++ memory model defines the memory access rules for a correct program.
  - Allows portable programming with lock free data structures.
- Atomic types allow to perform memory operations specifying an ordering.
  - Default ordering is sequential consistency.
- Relationships *synchronizes-with* and *happens-before* define constraints on operations ordering.
- Barriers allow to force orderings without modifying data.

References

- *C++ Concurrency in Action. Practical multithreading.*
  Anthony Williams.
  Chapter 5.

Memory consistency in C++

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Francisco Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid