C++ - parallelization and synchronization

Memory models, atomics, lock-free structures

Tomáš Faltín

Content

- Memory models
  - Memory ordering
  - Fence
- Atomics
- Lock-free structures
  - Problems

Memory models

- What is it?
  - What types of memory ordering you may expect for a given CPU or toolchain
- Weak
  - Really weak
    - Any load and store operations can be reordered with any other load and store, as long as it would never modify the behavior of a single thread
    - DEC Alpha, C++11
  - Weak with data dependency ordering
    - If you write A to B, then loading B is as new as A
    - ARM, PowerPC, SPARC (older models), IA-64 (Itanium)
- Strong
  - Every instruction comes with acquire and release semantics
  - When one CPU performs a sequence of writes, every other CPU sees those value change in the same order
  - Usually
    - In certain cases the strong ordering is lost
    - x86/64, SPARC in TSO
  - Sequential consistency
    - No memory ordering
    - No HW these days, only SW memory model (Java, C++)

Memory model

- Demo memory model for AMD64
  - Single processor
    - Out-of-order R are allowed
    - Speculative R are allowed
    - R can be reordered ahead of W
    - R cannot be reordered ahead of W if the R is from the same location as the prior W
      - Stall the instruction until W completed
    - Instruction fetch is parallel, asynchronous stream of R that is independent and unordered with loads from instructions
    - Out-of-order W are not allowed
    - Speculative W are not allowed
    - Write buffering is allowed
    - Write combining is allowed

Memory model

- Demo memory model for AMD64
  - Multiprocessor
    1. All L, S from a single CPU appear in program order
    2. Successive S from a single CPU are committed to memory and visible to other CPUs in program order
    3. S from a CPU cannot be reordered prior L
    4. S can be delayed by buffering, therefore S from s CPU may not appear to be sequentially consistent
    5. Non-overlapping L may pass S
    6. Dependent S between different CPUs appear to occur in program order
    7. Local visibility may differ from the global visibility (data bypass)

Memory model - Examples

- R1: All L, S from a single CPU appear in program order
  - L A cannot R 0 when L B reads 1

CPU0	CPU1
S A=1	L B
S B=1	L A

- R3: S from a CPU cannot be reordered prior L
  - L A and L B cannot both read 1

CPU0	CPU 1
L A	L B
S B=1	S A=1

- R4: S can be delayed by buffering, therefore S from s CPU may not appear to be sequentially consistent
  - Both L A and L B may read 1

CPU0	CPU1
S A=1	S B=1
…	…
S A=2	S B=2
L B	L A

Memory model – Examples

- R5: Non-overlapping L may pass S
  - All combinations (00, 01, 10, 11) may be observed by both CPUs

CPU0	CPU1
S A=1	S B=1
L B	L A

- R6: Dependent S between different CPUs appear to occur in program order
  - If CPU1 reads a value from A before carrying out S B, and if CPU2 reads updated value from B, a subsequent R A must also be updated value

CPU0	CPU1	CPU2
S A=1		
	L A (1)	
	S B=1	
		L B (1)
		L A (1)

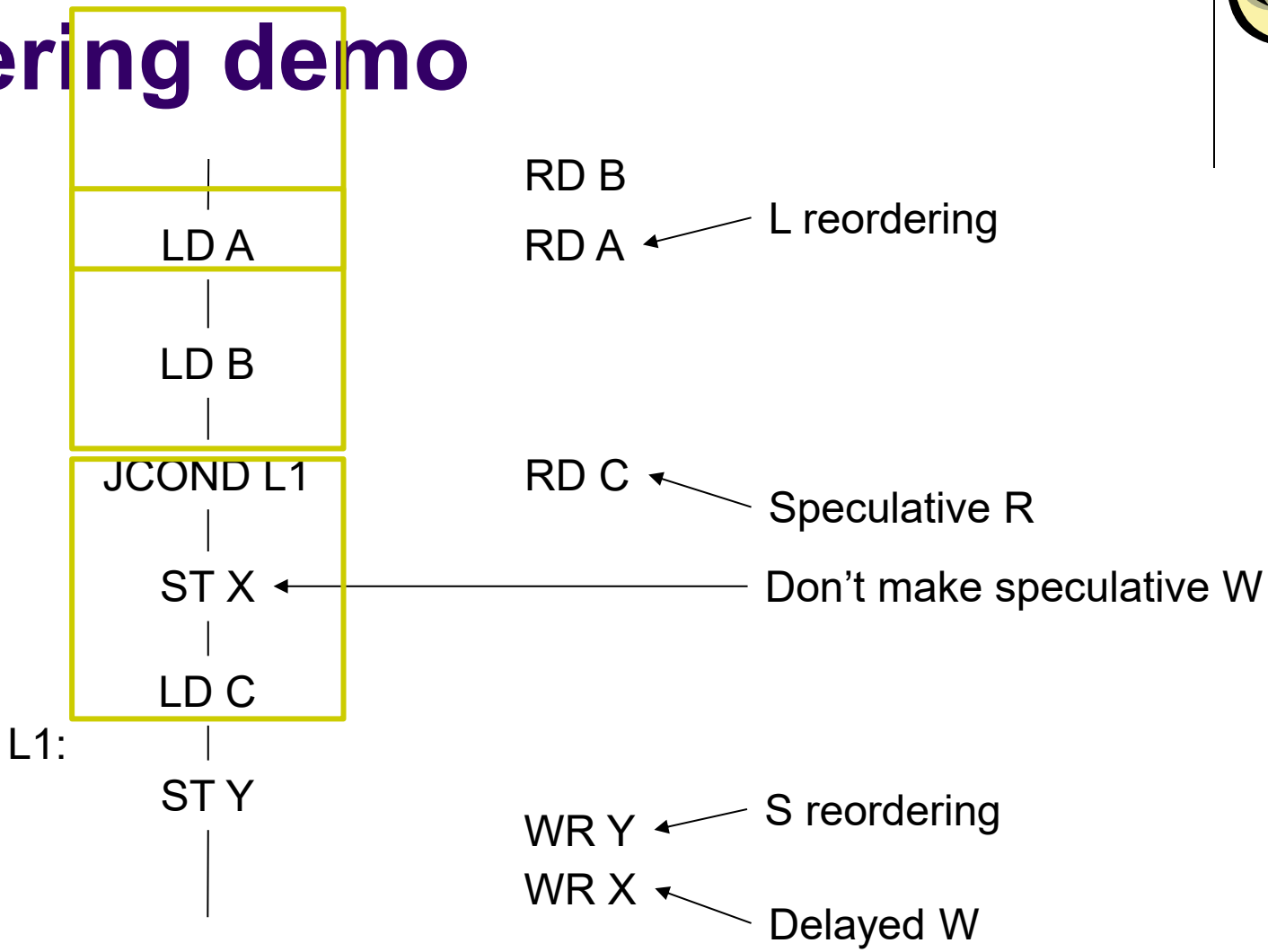
- R7: Local visibility may differ from the global visibility (data bypass)
  - L A in CPU0 can read 1 using data bypass, while L A in CPU1 can read 0, similarly for B

CPU0	CPU1
S A=1	S B=1
L A	L B
L B	L A

Memory models

- Barriers
  - Acquire barrier
    - All loads read after acquire will perform after it (loads do not overtake acquire)
  - Release barrier
    - All stores written before release are committed before the release (writes do not delay)

Memory model – totally weak ordering demo



Memory model – barrier demo

CPU0

CPU1

ST A

BARRIER

RD A

BARRIER

LD A

WR A

WR A

ST.REL A

BARRIER

BARRIER

LD.ACQ A

RD A

Atomic operations

- Atomic operations
  - Header <atomics>
  - Allows creating portable lock-free algorithms and data structures
  - Memory ordering
  - Fences
  - Lock-free operations, algorithms, data-structures

Atomic operations

- Memory ordering - `enum memory_order;`
  - `memory_order_seq_cst`
    - Sequentially consistent, most restrictive memory model
  - `memory_order_relaxed`
    - Totally relaxed memory model, allows best freedom for CPU and compiler optimizations
  - `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`
    - Additional barriers, weaker then sequentially consistent, stronger then relaxed

Atomic operations

- Easy way to make the demo safe

```
#include <atomic>

struct Counter {
  std::atomic<int> value;
  void increment(){ ++value; }
  void decrement(){ --value; }
  int get(){ return value.load(); }
};
```

Atomic operations

- Template atomic
  - Defined for any type
    - Load, store, compare_exchange
  - Specialized for bool, all integral types, and pointers
    - Load, store, compare_exchange
    - Arithmetic and bitwise operations
      - fetch_add
  - Wait, notify
    - Wait on atomic value change
    - Unblock waiting thread(s)

Atomic operations

- Atomic flag
  - `atomic_flag` allows one-bit test and set
- Atomic operations for `shared_ptr`

Atomic operations

- Fences
  - Explicit memory barrier
  - `void atomic_thread_fence(memory_order order) noexcept;`
    - `memory_order_relaxed`
      - No effect
    - `memory_order_acquire`
      - An acquire fence
    - `memory_order_release`
      - A release fence
    - `memory_order_acq_rel`
      - Both an acquire and a release fence
    - `memory_order_seq_cst`
      - Sequentially consistent

Busy Loop - Example

```
// shared variables
bool ready = false;
int data = 0;
```

```
// Thread 0 (producer)
data = 42;
ready = true;
```

No ordering guarantee

```
// Thread 1 (consumer)
while (!ready) {
    // busy loop
}

std::cout << data << "\n";
```

Busy Loop - Example

```
// shared variables
std::atomic_flag ready = ATOMIC_FLAG_INIT;
int data = 0;
```

```
// Thread 0 (producer)
data = 42;
ready.test_and_set(
  std::memory_order_relaxed);
```

No ordering guarantee

Guarantees **atomicity**

```
// Thread 1 (consumer)
while (!ready.test(
  std::memory_order_relaxed)) {
    // busy loop
}

std::cout << data << "\n";
```

Busy Loop - Example

```
// shared variables
std::atomic_flag ready = ATOMIC_FLAG_INIT;
int data = 0;
```

```
// Thread 0 (producer)
data = 42;
ready.test_and_set(
  std::memory_order_release);
```

```
// Thread 1 (consumer)
while (!ready.test(
std::memory_order_acquire)) {
    // busy loop
}
```

Release-acquire ordering

No reads or writes reordered **after** this **store**.

All writes are visible when acquiring the same atomic

No reads or writes reordered **before** this **load**.

Busy Loop - Example

```
// shared variables
std::atomic_flag ready = ATOMIC_FLAG_INIT;
int data = 0;


// Thread 0 (producer)
data = 42;
ready.test_and_set();
```

```
// Thread 1 (consumer)
while (!ready.test()) {
  // busy loop
}

cout << data << "\n";
```

Sequentially-consistent ordering (default)

Loads acquire, Stores release
Single total modification order

Single Total Modification Order - Example

```
// shared variables
std::atomic<int> x{0}, y{0};

void writer() {
  x.store(1, std::memory_order_relaxed);
  y.store(1, std::memory_order_relaxed);
}

void reader() {
  int xval = x.load(std::memory_order_relaxed);
  int yval = y.load(std::memory_order_relaxed);
  std::print("{} {}", xval, yval);
}
```

What if we change MO to **SEQ_CST**?

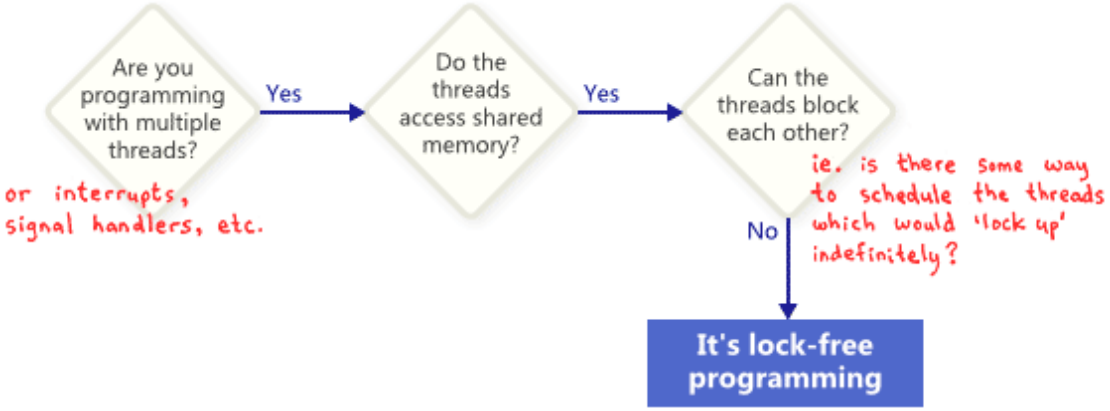
What are the possible outputs for 2 readers?

Mutex - Example

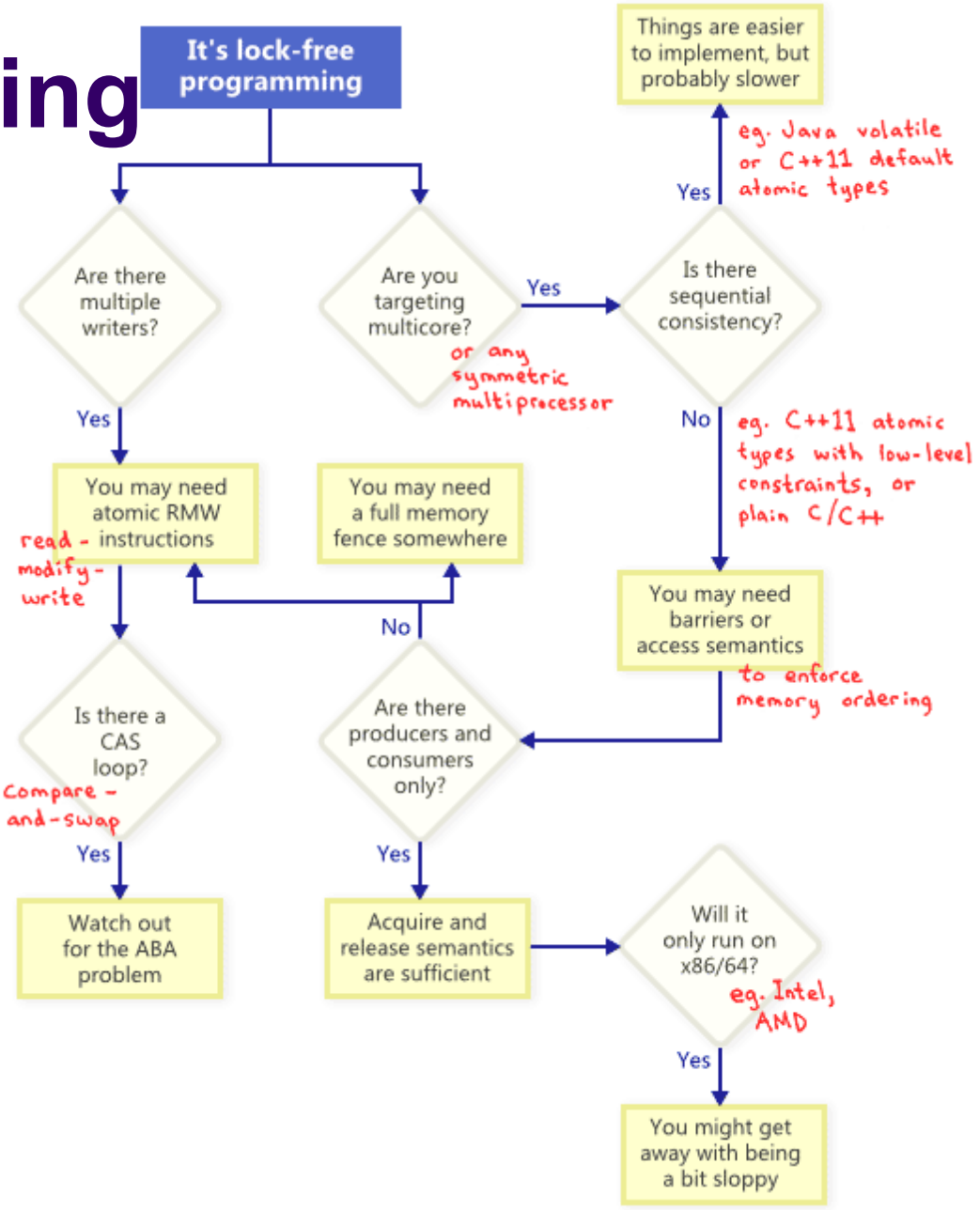
```
class mutex {
  std::atomic_flag m_{};
public:
  void lock() noexcept {
    while (m_.test_and_set(std::memory_order_acquire))
      m_.wait(true, std::memory_order_relaxed);
  }

  bool try_lock() noexcept {
    return !m_.test_and_set(std::memory_order_acquire);
  }

  void unlock() noexcept {
    m_.clear(std::memory_order_release);
    m_.notify_one();
  }
};
```

```
test_and_set() + clear()
wait() + notify_one()
```

Lock-free programming



Lock-free programming



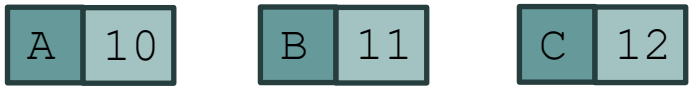
CAS loop - Example

```
for(;;) {
   // read critical data to a local var
   list_node *old_head = head;
   // speculatively modify new data
   pushed_member->next = old_head;
   // CAS - attempt to write critical data
   if(CAS(&head, pushed_member, old_head)==
      old_head) return;
}
```

atomic::compare_exchange_weak
atomic::compare_exchange_strong

ABA problem

- Problem in lock-free programming
  - Compared data looks same but they are not
  - Workarounds
    - Tag, counter
      - Use free bits, wraparound
      - Large CAS
        - All modern architectures have 128-bit CAS
    - Intermediate nodes
      - Expensive, not using pointers directly
    - Deferred reclamation



Unused Slides