Modern C/C++ concurrency

Torvald Riegel

2015/08/09

Focus of this tutorial

- On-topic
  - Managing complexity associated with concurrency
  - Getting more confident when working on concurrent code
  - Communicating about complex concurrent code
  - Shared-memory synchronization in multi-threaded C/C++ programs
- Off-topic
  - Specific synchronization / optimization techniques (except examples)
  - Performance considerations
  - Large distributed systems with failures
  - Why I'm making the specific recommendations in this tutorial
    - Ask me later!

Agenda

- Foundations
  - Terminology, concepts, …
  - Mental model for reasoning, communication, documentation
- C11/C++11 memory model
- Understanding and designing concurrent code
- Communicating about concurrency

- Not a (code-)example-driven tutorial, deliberately
- Extend and continue after the break?

Foundations

What is concurrency?

- Things “happen at the same time”
  - No simple ordering
- But there always is an ordering at some level!
  - Indivisible operations exist at some level (e.g., cache coherence states)
  - We're still thinking about state machines...
- However, not necessarily a strict total order!
  - Not necessarily a sequence – like sequential code
- Asynchronous systems – timing of individual operations unknown
  - For shared memory synchronization, sufficient to assume no failures: Time required to execute an operation is unknown but finite

Executions of concurrent systems

- Execution of a concurrent system is an execution of indivisible *steps*
    - Steps have sequential specifications and are atomic
    - Threads/processes all execute steps
        - Steps by one thread are totally ordered (per-thread program order)
    - Steps are ordered by a *happens-before* relation
    - For now, sufficient to assume that all executions have a strict-totally ordered happens-before
        - Not true for all of the C/C++ memory model…

Reasoning about concurrent systems

- Reasoning about one execution vs. reasoning about a system that can be executed
- Concurrent systems are asynchronous systems: All permutations of steps are possible unless prohibited by the concurrent algorithm

- Concurrent algorithms allow <u>sets</u> of possible executions
    - Need to reason about <u>all</u> of them!
    - Perhaps think about state space exploration trees, or other graphs

- That's the main source of complexity, and what we need to tackle!

Concurrent algorithms: Two goals

- Safety guarantee: <u>Nothing bad ever</u> happens
  - No executions possible that don't satisfy the algorithm's requirements
  - Essentially, algorithm prevents bad executions by constraining executions

- Liveness(/progress) guarantee: <u>Something good eventually</u> happens
  - In practice, often simpler than safety
  - Remember that we assume that each thread will execute a step eventually

- Think about both, separately!

Constraining possible executions: Atomicity

- Atomic = indivisible in the execution
  - But always wrt certain other steps – document which unless obvious!
  - Atomicity does <u>not</u> necessarily imply global total order! (eg, only atomic wrt X)

- Examples of means to achieve atomicity
  - Atomic store to memory: globally atomic (at level of C/C++ implementation)
  - Atomic read-modify-write such as CAS: globally atomic
  - Locks: critical sections atomic wrt other critical sections using the same lock
  - Thread-private data: not accessible to other threads

- Benefits of atomicity: Steps of a thread become "bigger"
  - Larger granularity leads to fewer possible executions

Constraining possible executions: Ordering constraints

- Constrain ordering by adding/enforcing edges to/in happens-before relation
  - Prohibit all executions that do not have such an edge (IOW, have badly ordered steps)

- Examples of means to constrain ordering
  - “Wait” for something to happen
  - Several “base rules” in the C11/C++11 memory model
  - C11/C++11 memory orders

- Benefits of ordering constraints: Fewer permutations of steps allowed

C11/C++11 memory model

What is a memory model?

- Definition of how memory behaves when accessed by multiple threads
- For shared-memory synchronization, this is the base level defining steps!
  - Individual steps of threads are Other Stuff ending with a memory access

- Hardware-level memory models
  - Arch-specific, reasoning at HW instruction level
  - Model effects of caches and cache coherency protocols

- Programming-language-level memory models
  - Define basic abstractions (e.g., what is a “memory location”?)
  - Capture effects of HW memory models (portably!)
  - Model effects of **compiler optimizations!**

How the C11/C++11 model works

- Different approaches to understanding the model – we'll follow the formalization of the model by Batty et al.
- *Memory locations* as defined by the C/C++
- Start with possible control flows of the abstract machine
  - Actions performed on such paths: reads, writes, locks, unlocks, fences
- Then enumerate possible choices for each action
  - Which store a load reads from
  - How the value of each memory location changes during the execution
  - Ordering of actions requested to be sequentially consistent
- Result: A set of candidate executions
  - For each of these, derive additional relations (e.g., *happens-before*)
- Ignore all non-consistent candidate executions
  - Inconsistent ones are an implementation detail of the model's formalization
- <u>All</u> consistent candidate executions must be data-race-free, or UB
- Implementation can pick <u>any</u> of the consistent candidate executions

Data-race-freedom (DRF) requirement

- Goal: Distinguish between concurrent code and sequential code
  - Compiler can optimize sequential code <u>much</u> more effectively
  - Separation of atomically accessed data and types (and locks) from non-atomic-typed data
- Data race exists if there is a pair of memory accesses, and
  - Both are to the same memory location, and
  - At least one is a write, and
  - At least one is not atomic, and
  - The accesses are not ordered by happens-before
- If <u>any</u> of the candidate executions has a data race, then the whole program has undefined behavior (not just this execution!)
  - Undefined behavior can mean that compiler generates garbage…
- Compiler must not introduce data races into a DRF program!

The cppmem tool

- Finds all possible executions of multi-threaded C/C++ synchronization
  - Uses formal model of C++11 memory model to find all candidate executions, detects which are consistent and whether they have data races
  - Runs in your browser – good enough for small examples
- http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/
- Supports reduced C-like syntax:
  - Variables: int and atomic_int
  - Thread-local variables: don't declare, just use r<N> as identifier
  - Threads: {{{ { <code thread 1> } ||| … ||| { <code thread N> } }}}
  - Access to non-atomic variables: like in C
  - Atomic operations:
    - r1 = a.load(memory_order_relaxed); b.store(1, memory_order_release);
    - r2 = c.load(memory_order_acquire).readsvalue(42);
    - if (cas_weak(a, expected, desired)) …;
    - atomic_thread_fence(memory_order_release);

sequenced-before

- Per-thread program order
  - One full expression sequenced-before the next one
  - Not all evaluations of a thread are sequenced – see C++14, 1.9p13-15
- sequenced-before is part of happens-before
  - happens-before includes further relations as explained later

Example

- Simple cppmem program that creates two threads – Try it!
  - Use top-left text field to provide program (e.g., example below)
  - Options on bottom-left control which relations are shown in visualization of the each execution at the bottom-right (e.g., “Wna” is non-atomic write)
  - Top-right buttons navigate through all (consistent) executions, true/false flags show which rules are satisfied or not for consistency/DRF
- Observe sequenced-before (sb), happens-before (hb)

```
int main() {
  int x = 0;
  int y = 0;
  {{{ { x = 3; }
  ||| { y = 1;
        y = 2; }
  }}};
  return 0; }
```

modification-order and reads-from

- *modification-order*: Strict total order of all modifications for each memory location (i.e., one order per location)

- *reads-from*: Which executed store a load reads its value from
- Is effectively constrained by happens-before
  - Every write action that happens-before the read with no intervening write is a *visible side effect*
  - Non-atomic actions must read-from visible side effect
  - Atomic actions must read-from visible side effect or subsequent write in modification order
  - Atomic read-modify-write (e.g., CAS) read from last preceding write in modification-order
  - If this does not hold, candidate execution is not consistent

Example

- Now we add atomics
- Observe modification-order (mo), reads-from (rf)

```
int main() {
  int x = 0;
  atomic_int y = 0;
  {{{ { x = 3; }
  ||| { y.store(1);
        r1 = y.load(); }
  }}};
  return 0; }
```

Sequentially consistent atomic accesses

- Sequential consistency: Operations are in a strict total order that is consistent with per-thread program order
- In the model:
  - *sc*: strict total order of all sequentially consistent operations
  - By default, atomic memory accesess are sequentially consistent
  - sc must be consistent with both happens-before and modification-order
    - i.e., happens-before restricted to sc must be subset of sc
  - If read is in sc, and it reads-from an sc write, then the write must be the most recent write to this memory location in sc
  - Otherwise, candidate execution is inconsistent

Example

- Now 3 consistent executions
    - Use top-right buttons to look at all three
- Observe modification-order (mo), reads-from (rf), sc
- Observe consistency between those relations
    - E.g., edges in sc go same direction as hb edges

```
int main() {
  atomic_int y = 0;
  {{{ { y.store(3); }
  ||| { y.store(1);
        r1 = y.load(); }
  }}};
  return 0; }
```

Example

- Use readsvalue() to just look at certain executions

```
int main() {
  atomic_int y = 0;
  {{{ { y.store(3); }
  ||| { y.store(1);
        r1 = y.load().readsvalue(1); }
  }}};
  return 0; }
```

Locks

- Behave as expected
- Lock/unlock actions form strict total order per lock instance
  - Lock/unlock alternate
- Lock/unlock action considered to be part of sc
  - Thus, must be consistent with happens-before and modification-order
  - Thus, also effectively constrain reads-from

Explicit memory orders

- Express different ordering constraints
  - Strong ordering constraints are less efficient
- memory_order_{relaxed,acquire,release,acq_rel,seq_cst}
  - Don't use memory_order_consume
  - seq_cst is strongest, relaxed weakest:

$$\begin{tikzcd}
& \text{acquire} \arrow[dr] & & \\
\text{relaxed} \arrow[ur] \arrow[dr] & & \text{acq\_rel} \arrow[r] & \text{seq\_cst} \\
& \text{release} \arrow[ur] & &
\end{tikzcd}$$

- Atomic loads, stores, RMW ops, and fences all take a memory order argument
  - CAS has two: one for successful CAS, one for when it fails
  - Side note: weak CAS can fail spuriously, strong CAS cannot

Example

- Now explicit memory orders – not sequentially consistent by default anymore
- Observe all 3 consistent executions

```
int main() {
  atomic_int y = 0;
  {{{ { y.store(3, memory_order_relaxed); }
  ||| { y.store(1, memory_order_relaxed);
        r1 = y.load(memory_order_relaxed); }
  }}};
  return 0; }
```

Coherence rules

- Did you notice we didn't read initial 0 in the previous example?
- Four rules guaranteeing the intuitive behavior:
  - CoRR: two reads in one thread must not read-from stores ordered opposite to modification-order
  - CoRW: a read sequenced-before write must not read-from another write later in modification-order
  - CoWR: if write sequenced-before read, then read must not read-from another write earlier in modification-order
    - That's why we don't read 0
  - CoWW: modification-order and happens-before must be consistent

synchronizes-with

- Release action: atomic write with memory_order_{release,acq_rel,seq_cst}
- Release action has *release sequence*:
    - Contiguous part of modification-order starting with release action
    - Each element of the sequence either (1) by same thread as release action or (2) atomic read-modify-write op
- Acquire action: atomic read with memory_order_{acquire,acq_rel,seq_cst}
- Acquire action *synchronizes-with* release action if the former reads-from a write in the release sequence of the latter
- Happens-before is transitive closure of union of synchronizes-with and sequenced-before
    - Consistent happens-before has no cycles
    - Standard / formal model is more complex (*inter-thread-happens-before*) due to memory_order_consume that we ignore here
    - Remember: happens-before decides whether reads-from is consistent

Example

- Observe two possible executions: first one doesn't load x, second does
- Observe sychronized-with (sw) and how it affects reads-from via happens-before
- Try changing at least one memory order to memory_order_relaxed and observe the resulting data race

```
int main() {
  int x = 2;
  atomic_int y = 0;
  {{{ { x = 1;
        y.store(1, memory_order_release); }
  ||| { if (y.load(memory_order_acquire))
          r1 = x; }
  }}};
  return 0; }
```

Example (from glibc TLS code)

- Now with additional relaxed RMW by a third thread
- We just look at the interesting case: acquire-read reads-from the RMW
- Observe release-sequence (rs) and synchronized-with (sw)

```
int main() {
  int x = 2;
  atomic_int y = 0;
  {{{ { x = 1;
        y.store(1, memory_order_release); }
  ||| { cas_weak_explicit(y,1,2, memory_order_relaxed,
                                 memory_order_relaxed); }
  ||| { y.load(memory_order_acquire).readsvalue(2);
        r1 = x; }
  }}};
  return 0; }
```

And that's it

- I omitted a few details (e.g., fences)
- memory_order_consume ignored because in practice it is implemented as memory_order_acquire
  - Specification in the standard is not practical to implement

- It's useful to be able to understand the model
- In practice, one will likely only remember certain patterns
  - For example, release/acquire pair happens-before
  - Use cppmem to look up what's allowed if not sure, or if dealing with a corner case

Reviewing and designing concurrent code

How do we apply all this in practice?

- Writing concurrent code (details: next slides)
  - Understand “input” concurrency
  - Understand requirements
  - Find a high-level synchronization scheme
  - Implement the synchronization scheme
  - Repeat / refine

- Review of concurrent code
  - Review all of the above
  - Or add documentation about it if not present

(1) Which concurrent executions are there, potentially?

- Examples:
  - Which functions can be called concurrently?
  - Are there atomicity or ordering constraints guaranteed by the callers of those functions?
- Result:
  - A broad understanding of the concurrency that is possible
  - This is the problem space we're dealing with
- Add to documentation!

(2) Which requirements do we have to satisfy?

- Safety guarantees
  - “Thread-safe” is typically not a sufficient requirement
  - Examples for better conditions:
    - All function calls behave as-if atomic and sequentially consistent
      - I.e., atomic and in a strict total order that's consistent with program order (sequenced-before) and happens-before
    - All uses of the data structure happen-before it's destroyed
- Liveness guarantees (often simpler)
  - Examples:
    - No deadlock (often an obvious requirement)
    - No starvation of individual threads (assuming randomized execution interleavings often okay)
- Add to documentation!

(3) Find a high-level synchronization scheme

- Reduce the set of (bad) executions using some scheme that still fits into your head
- Reduce diversion between threads
  - Consensus: all threads agree to next do (or assume) something – simpler because all on same page
  - Look for operations that create diversion
    - E.g., pending stores: thread already decided to store something, but hasn't yet
    - See bounds-better-than-examples recommendation later on
- Assign roles and responsibilities to threads or parts of the algorithm
  - Delegate work to certain threads
- Examples:
  - Use consensus to make one thread responsible for clean-up; all others wait for clean-up to finish
- Document it! If you can't describe it, perhaps something is wrong...

(4) Implement synchronization scheme

- Pick the right set of techniques / mechanisms!
- Using locks
  - Go from the high-level scheme to critical sections (via high-level atomicity and ordering)
  - Then derive detailed locking scheme: which data protected by which locks, which operations are atomic
  - Transactions: same as for single global lock
  - Document the locking scheme!
- Using atomics
  - Suggestion: Start with the key synchronization points in your algorithm: consensus points, responsibility delegation, …
  - Memory orders: Start with seq_cst, then reduce to what you really need.
    - Document them!: Why sufficient, why necessary

(5) Repeat/refine until all requirements satisfied

- Remember: No bad executions and eventually good ones
- Use as many levels of abstraction as you need to make it tractable
    - Fewer (bad) executions to consider = simpler problem
    - Within each level of abstraction, same approach
- Back-track if requirements can't be implemented
    - Happens quite a bit when designing more complicated concurrent algorithms
    - Don't get frustrated, just keep working systematically :)
- Use patterns you (and your fellow developers) understand to break down the complexity

Communicating about concurrency

Code documentation

- Be verbose!
- Make precise statements, no hand-waving!
- We have to reason about intervals/sets, not examples!
  - Clearly separate examples from statements that are tight
  - State precise bounds of sets of possible executions
    - We need to reason about all of them!
    - Bounds are not bounds, actually: we forget about executions
    - Bounds not tight: we don't reduce complexity as much as we could
- Use common terminology/patterns
  - There are correct and incorrect choices – but no one size fits all
  - Use something that works for the project and its developers
  - But make sure it has sound foundations
    - E.g., if using a simpler mental model, there must be a mapping to the C11/C++11 memory model

Why document the abstract algorithm?

- It's simpler than the implementation!
  - Higher level of abstraction
  - Often simpler than memory model details

- It represents the intent of the programmer
  - Reconstructing the intent from the implementation is very time-consuming and error-prone in case of concurrent code!
  - Can cross-check intent against implementation
  - Redundancy is useful when dealing with complicated matters!

Further reading

- Herlihy, Shavit: The Art of Multiprocessor Programming
  - Great text book about shared-memory synchronization
  - All the necessary foundations and theory
  - The major synchronization techniques
  - Uses Java, but general techniques and algorithms are the same
- Batty et al.: Mathematizing C++ Concurrency: The Post-Rapperswil Model
  - http://www.cl.cam.ac.uk/~mjb220/n3132.pdf
  - Describes formal model of C++11 memory model (base for the cppmem tool)
  - May look like lots of formulae, but IMO overall simpler to understand than the C++ standards prose
- glibc concurrency guidelines: https://sourceware.org/glibc/wiki/Concurrency
  - WIP

Backup Slides

Comparison of sync techniques

- Coarse-granular locking: Simple, might not scale well
  - Lock elision can make a difference
- Fine-granular locking: Can get complex, can scale well
- TM with HTM: Simple, performance really depends on workload
- TM w/o HTM: Simple, can scale well but higher single-thread overheads
- Atomics for simple patterns: Straight-forward, fast
  - Simple data hand-over
  - Simple consensus
  - Lock-like constructs
- Atomics for complex patterns: Complex, getting good performance can be complex too

ABA issues

- CAS checks equality of value to see whether to apply the change
- ABA issue
  - State representation changes from value A to value B and back to A
  - If first A actually represents a different logical state than the second A, you're in trouble
- Detection: do states that need to be distinguished are represented using different values?
- Workarounds:
  - Widen your representation (e.g., include a counter in state representation)
  - Quiesce phase of first A: Wait until all threads are aware of state B