C/C++11 Memory Model

Xiao Jia

Pygmal Technologies

Roadmap

- Compiler optimizations are correct only for single-threaded programs.
- Data race causes correctness issues.
- Lock causes performance issues.
- Trade-off:

  Correctness (no optimization at all) vs. Performance (optimize as much as possible).
- Root cause:

  Compiler doesn't know which variables are shared by multiple threads.
- Solution: Tell the compiler (by using std::atomic<>).
- Compiler can now safely optimize some of the multi-threaded programs.

Data race

- Two (or more) threads access the same memory location concurrently, and
- at least one of them is for writing.
- `x = 1; || y = x;`
- `x = 1; || x = 2;`
- What is a memory location?

Compiler optimizations are correct only for ST programs

- x = 1; r1 = y;
- No data dependency between x and y.
- Performing loads early may lead to performance improvements.
- Reordered: r1 = y; x = 1;
- y = 1; r2 = x;
- Reordered: r2 = x; y = 1;
- “Compile-time memory (re)ordering”
- Reordered statements in two threads lead to unintuitive outcome: r1 = r2 = 0.

Speculative execution

- if (cond) work();
- If it is known (e.g. by using profiling tools) that “cond” almost always evaluates to true
- and “work()” is undo-able,
- it could be optimized as
- work(); if (!cond) undo_work();
- if (x == 1) ++y; can be optimized as ++y; if (x != 1) --y;
- if (y == 1) ++x; can be optimized as ++x; if (y != 1) --x;
- Unexpected outcome: x = y = 1.

Rewriting adjacent data

- On a little-endian 32-bit machine:
- struct { int a:17; int b:15; } x;
- x.a = 42; is likely to be implemented as
- tmp = x; // (1) read both fields into a 32-bit register

  tmp &= ~0x1ffff; // (2) mask off old value of “a”

  tmp |= 42; // (3) fill in new value of “a” (i.e. 42)

  x = tmp; // (4) overwrite all of x
- A concurrent update to x.b between (1) and (4) may be lost.
- But the two threads are operating on completely distinct fields!

Lock causes performance issues

```
for (p = start; p < 10000; ++p)
  if (is_prime[p]) {
    for (x = p; x < 100000000; x += p)
      if (is_prime[x])
        is_prime.reset(x);
  }
```

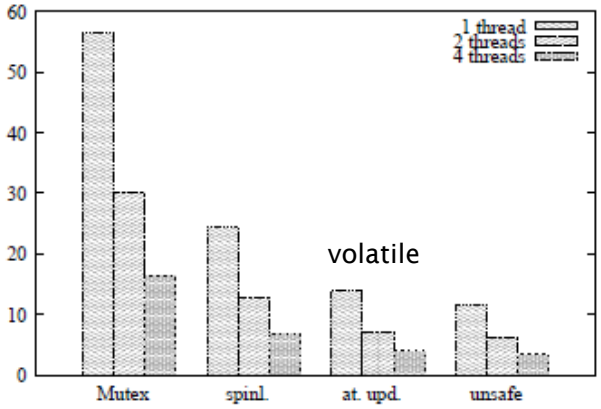


Figure 2: Sieve execution time for bit array (secs)

Introducing std::atomic<>

- #include <atomic>
- std::atomic<int> x, y;
- x.store(1); r1 = y.load();
- y.store(1); r2 = x.load();
- // or simply
- x = 1; r1 = y;
- y = 1; r2 = x;
- Loads and stores are sequentially consistent by default and always recommended.
  - the result of any execution is the same as if
    the operations of all the processors were executed in some sequential order, and
    the operations of each individual processor appear in this sequence in the order specified by its program

Release-acquire ordering

- Thread A: `x.store(1, std::memory_order_release);`
- Thread B: `y = x.load(std::memory_order_acquire);`
- Once the load is completed, B is guaranteed to see everything A wrote to memory.

Release-acquire ordering

- ```
  std::atomic<string*> ptr;
  int data;  // Note: this is not an atomic variable.
  ```
- ```
  void producer() {
        string* p1 = new string(“Hello”);
        data = 42;
        ptr.store(p1, memory_order_release);
  }
  ```
- ```
  void consumer() {
        string* p2;
        while (!(p2 = ptr.load(memory_order_acquire))) /* spin */;
        assert(*p2 == “Hello”);
        assert(data == 42);
  }
  ```
  ```

Relaxed ordering

- std::atomic<int> x, y;
- x.store(1, memory_order_relaxed);
  r1 = y.load(memory_order_relaxed);
- y.store(1, memory_order_relaxed);
  r2 = x.load(memory_order_relaxed);
- Valid outcome: r1 = r2 = 0

Simple Event Counting

- Consider (*count* is atomic, initially zero):
  - Threads **1..N**: Incrementing.

```
while( ... ) {
   :::
   if( ... )
    ++count;
   :::
}
```

Main thread.

```
int main() {
   launch_workers();
   :::

   join_workers();
   cout
      << count
      << endl;
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - Hint: Thread exit *happens-before* returning from a join with that thread.

Simple Event Counting

- Consider (*count* is atomic, initially zero):
  - Threads **1..N**: Incrementing.

```
while( ... ) {
   :::
   if( ... )
    count.fetch_add(1,memory_order_relaxed);
   :::
}
```

Main thread.

```
int main() {
   launch_workers();
   :::

   join_workers();
   cout
      << count.load(memory_order_relaxed)
      << endl;
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - A: *count* incs/stores can be relaxed – **it is not part of the comm between threads.**

Simple Event Counting: Better Solution

- Consider (*count* is ***event_counter***, initially zero):
  - Threads **1..N**: Incrementing.

```
while( ... ) {
   :::
   if( ... )
    ++count;
   :::

}
```

Main thread.

```
int main() {
   launch_workers();
   :::

   join_workers();
   cout
      << count
      << endl;
}
```

- Better: Use a type that encapsulates the desired semantics and hides the relaxed memory ops.

Simple Flag Setting

- Consider (*dirty* and *stop* are atomic, initially false):
  - Threads **1..N**: Dirty setting.

```
while( !stop ) {
   if( ::: )
      dirty = true;
   :::

}
```

Main thread.

```
int main() {
   launch_workers();
   stop = true;
   join_workers();
   if( dirty )
      clean_up_dirty_stuff();
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - Hint: Thread exit *happens-before* returning from a join with that thread.

Simple Flag Setting

- Consider (*dirty* and *stop* are atomic, initially false):
  - Threads **1..N**: Dirty setting.

```
while(!stop.load(memory_order_relaxed)) {
  if( ::: )
    dirty.store(true,memory_order_relaxed);
  :::
}
```

Main thread.

```
int main() {
  launch_workers();
  stop = true;        // not relaxed
  join_workers();
  if( dirty.load(memory_order_relaxed) )
    clean_up_dirty_stuff();
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - *dirty* can be relaxed, relying on "join"'s ordering (doesn't itself publish data).
  - *stop.load* can be relaxed <u>if setting *stop* doesn't publish data</u>.

Simple Flag Setting

- Consider (*dirty* and *stop* are atomic, initially false):
  - Threads **1..N**: Dirty setting.

```
while(!stop.load(memory_order_relaxed)) {
  if( ::: )
    dirty.store(true,memory_order_relaxed);
  :::
}
```

Main thread.

```
int main() {
  launch_workers();
  stop = true;        // not relaxed
  join_workers();
  if( dirty.load(memory_order_relaxed) )
    clean_up_dirty_stuff();
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - *dirty* can be relaxed, relying on "join"'s ordering (doesn't itself publish data).
  - *stop.load* can be relaxed <u>if setting *stop* doesn't publish data</u>.
- **Q2: Is it worth it?**

Simple Flag Setting: Better Solution

- Consider (*dirty* and *stop* are ***dirty_flag***, initially false):
  - Threads **1..N**: Dirty setting.

```
while( !stop ) {
   if( ::: )
      dirty = true;
   :::
}
```

Main thread.

```
int main() {
   launch_workers();
   stop = true;
   join_workers();
   if( dirty )
      clean_up_dirty_stuff();
}
```

- Better: Use a type that encapsulates the desired semantics and hides the relaxed memory ops.

Reference Counting

- Consider (*refs* atomic):
  - Thread 1: Increment.
    (inside, say, *smart_ptr* copy ctor)

```
:::
control_block_ptr
   = other->control_block_ptr;
++control_block_ptr->refs;

:::
```

Thread 2: Decrement.
(inside, say, *smart_ptr* dtor)

```
:::
if( --control_block_ptr->refs == 0 )
{
   delete control_block_ptr;

:::
```

- **Q: State exactly what ordering is needed on each atomic load and store.**

Reference Counting

- Consider (*refs* atomic):
  - Thread 1: Increment.
    (inside, say, *smart_ptr* copy ctor)

```
:::
control_block_ptr
   = other->control_block_ptr;
control_block_ptr->refs
   .fetch_add(1,memory_order_relaxed);
:::
```

  - Thread 2: Decrement.
    (inside, say, *smart_ptr* dtor)

```
:::
if( control_block_ptr->refs
      .fetch_sub(1,memory_order_acq_rel) == 0 ) {
   delete control_block_ptr;
:::
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
- **A:** Increment can be **relaxed** (not a publish operation).
  Decrement can be **acq_rel** (both acq+rel necessary, probably sufficient).

Why Not _*release*?

- Now let's look at two threads who are the last to leave this object:
  - Thread 1: Decrement 2->1.

```
// A: use object
if( control_block_ptr->refs
      .fetch_sub(1,memory_order_release)
      == 0 ) {
   // branch not taken
}
:::
```

  - Thread 2: Decrement 1->0.

```
:::
if( control_block_ptr->refs
      .fetch_sub(1,memory_order_release)
      == 0 ) {
   delete control_block_ptr;      // B
}
:::
```

- **Q: Is this code correct?**

Why Not _*release*?

- Now let's look at two threads who are the last to leave this object:
  - Thread 1: Decrement 2->1.

```
// A: use object
if( control_block_ptr->refs
      .fetch_sub(1,memory_order_release)
      == 0 ) {
   // branch not taken
}
:::
```

  - Thread 2: Decrement 1->0.

```
:::
if( control_block_ptr->refs
      .fetch_sub(1,memory_order_release)
      == 0 ) {
   delete control_block_ptr;      // B
}
:::
```

  - No acquire/release ⇒ no coherent communication guarantee that thread 2 sees thread 1's writes in the right order. *To thread 2*, line A could appear to move below thread 1's decrement even though it's a release(!).
  - Release doesn't keep line B below decrement in thread 2.

Reference Counting: Better Solution

- Consider (*refs* is ***atomic_ref_count***):
  - Thread 1: Increment.
    (inside, say, *smart_ptr* copy ctor)

```
:::
control_block_ptr
   = other->control_block_ptr;
++control_block_ptr->refs;

:::
```

  - Thread 2: Decrement.
    (inside, say, *smart_ptr* dtor)

```
:::
if( --control_block_ptr->refs == 0 )
{
   delete control_block_ptr;

:::
```

- Better: Use a type that encapsulates the desired semantics and hides the relaxed memory ops.

Traditional Double-Checked Locking

- The Double-Checked Locking (DCL) pattern is **no longer** broken.
  - Using C++11 notation:

```
atomic<widget*> widget::instance = nullptr;
widget* widget::get_instance() {
  if( instance == nullptr ) {                  // 1: first check (ATOMIC)
    lock_guard<mutex> lock( mutW );            // 2: THEN acquire lock (crit sec enter)
     if( instance == nullptr ) {               // 3: THEN second check (ATOMIC)
       instance = new widget();                // 4: THEN create, THEN assign (ATOMIC)
     }
  }                                            // 5: release lock (crit sec exit)
  return instance;                             // 6: return pointer
}
```

- Key steps involve both **atomicity** and **ordering**.

A Variant

- Alternative lazy initialization strategy.

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;
widget* widget::get_instance() {
  if( instance.load() == nullptr ) {
    if( ! create.exchange(true) )
      instance = new widget();                 // construct
    else while( instance.load() == nullptr ) { }   // or spin
  }
  return instance;
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - Hint: The fast case must perform at least a load-acquire of instance.

Option 1: Relaxed *exchange*?

- What if we make the *exchange* relaxed?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {          // _acquire
    if( ! create.exchange_explicit(true,memory_order_relaxed) )   // _relaxed (?)
      instance.store(new widget(),memory_order_release);          // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { } // _acquire
  }
  return instance.load(memory_order_acquire);                     // _acquire
}
```

- **Q: Is this correct?**

Option 1: Relaxed *exchange*?

- What if we make the *exchange* relaxed?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {          // _acquire
    if( ! create.exchange_explicit(true,memory_order_relaxed) )   // _relaxed (?)
      instance.store(new widget(),memory_order_release);          // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { } // _acquire
  }
  return instance.load(memory_order_acquire);                     // _acquire
}
```

- **A: No;** e.g., could do some widget creation even if CAS fails – and worse.

Option 2: Acquire *exchange*?

- What if we make the *exchange* acquire?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {          // _acquire
    if( ! create.exchange_explicit(true,memory_order_acquire) )   // _acquire (?)
      instance.store(new widget(),memory_order_release);          // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { } // _acquire
  }
  return instance.load(memory_order_acquire);                     // _acquire
}
```

- **Q: Is this correct?**

Option 2: Acquire *exchange*?

- What if we make the *exchange* acquire?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {          // _acquire
    if( ! create.exchange_explicit(true,memory_order_acquire) )   // _acquire (?)
      instance.store(new widget(),memory_order_release);          // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { } // _acquire
  }
  return instance.load(memory_order_acquire);                     // _acquire
}
```


- **A: Yes, but** there seems to be no benefit – same legal reorderings.

Option 3: Make Final Store Relaxed?

- What if we notice the final store is redundant, and fix it?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {          // _acquire
    if( ! create.exchange_explicit(true,memory_order_seq_cst) )   // _seq_cst
      instance.store(new widget(),memory_order_release);          // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { } // _acquire
  }
  return instance.load(memory_order_relaxed);                     // _relaxed (?)
}
```

- **Q: Is this correct?**

Option 3: Make Final Store Relaxed?

- What if we notice the final store is redundant, and fix it?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance == null                                            // _acquire
    if( ! create.exchange(true) )                                 // _seq_cst
      instance = new                                              // _release
    else while( instance == n                                     // _acquire
  }
  retur instance;                                                 // _acquire
}
```

s/instance/(**temp**=instance)/

s/instance/instance=**temp**/

s/instance/(**temp**=instance)/

s/instance/**temp**/

- A: Yes, but no benefit– compiler/HW can optimize redundant load anyway.

Formal reasoning about the C11 weak memory model

Invited talk @ CPP'15

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

13 January 2015

Sequential consistency

Sequential consistency (SC):

- The standard model for concurrency.
- Interleave each thread's atomic accesses.
- Almost all verification work assumes it.

Initially, $X = Y = 0$.

$$\begin{array}{l|l} X := 1; & Y := 1; \\ a := Y & b := X \end{array}$$

In SC, this program cannot return $a = b = 0$.

A basic guarantee: coherence

Coherence:

"SC for a single variable"

Initially, $X = 0$.

$$X := 1 \;\Big\|\; X := 2 \;\Big\|\; \begin{array}{l} a := X; \\ b := X \end{array} \;\Big\|\; \begin{array}{l} c := X; \\ d := X \end{array}$$

Forbidden outcome: $a = 1, b = 2, c = 2, d = 1$.

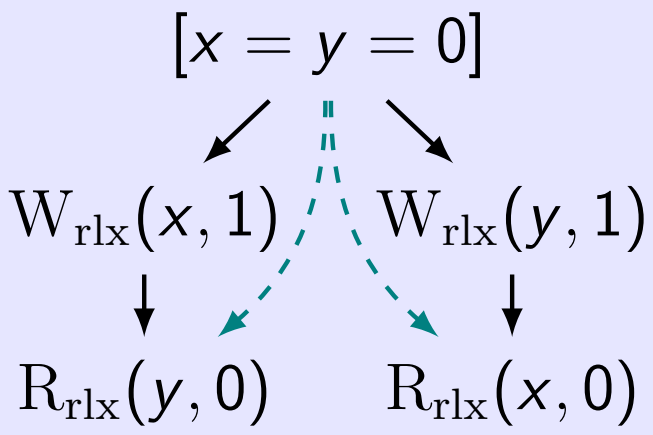
Relaxed behaviour: store buffering

Initially $x = y = 0$.

$$\begin{array}{l|l} x.\text{store}(1, \mathit{rlx}); & y.\text{store}(1, \mathit{rlx}); \\ t_1 = y.\text{load}(\mathit{rlx}); & t_2 = x.\text{load}(\mathit{rlx}); \end{array}$$

This can return $t_1 = t_2 = 0$.

Justification:



Behaviour observed on x86/Power/ARM

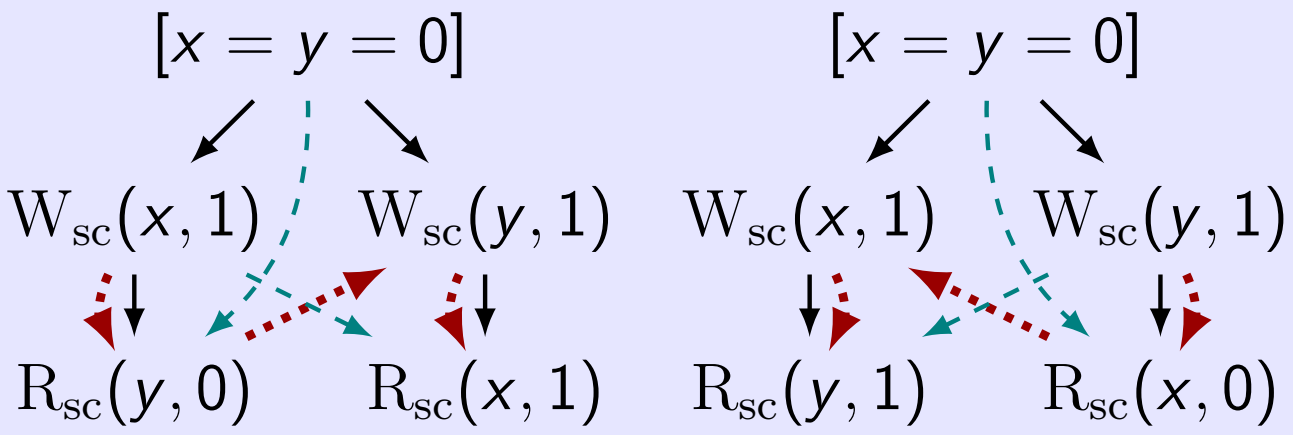
Getting rid of the SB behaviour

Initially $x = y = 0$.

$$\begin{array}{l|l} x.\mathsf{store}(1, sc); & y.\mathsf{store}(1, sc); \\ t_1 = y.\mathsf{load}(sc); & t_2 = x.\mathsf{load}(sc); \end{array}$$

This cannot return $t_1 = t_2 = 0$.

Justification:



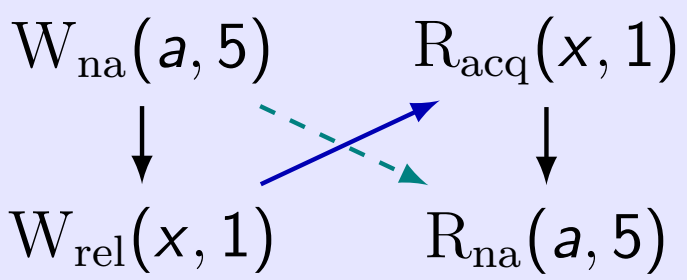
Release-acquire synchronization: message passing

Initially $a = x = 0$.

$a = 5;$	**while** $(x.\text{load}(acq) == 0);$
$x.\text{store}(1, \textit{release});$	$\text{print}(a);$

This will always print 5.

Justification:



Release-acquire synchronization

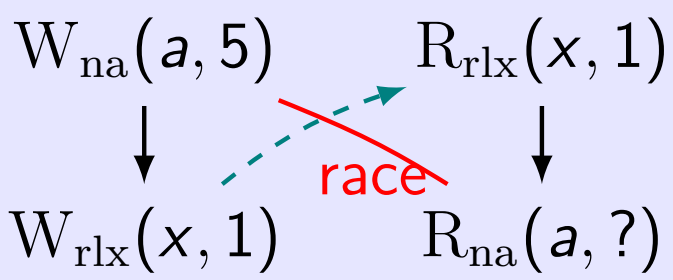
Relaxed accesses don't synchronize

Initially $a = x = 0$.

$a = 5;$	**while** $(x.\text{load}(rlx) == 0);$
$x.\text{store}(1, rlx);$	$\text{print}(a);$

The program is racy $\rightsquigarrow$ undefined semantics.

Justification:



Relaxed accesses don't synchronize

Understanding C11 using relaxed program logics

Separation logic

Key concept of *ownership* :

- Resourceful reading of Hoare triples.

$$\{P\}\ C\ \{Q\}$$

- Disjoint parallelism:

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}$$

Separation logic rules for non-atomic accesses

- Allocation gives you permission to access x.

$$\{\mathsf{emp}\}\ x = \mathsf{alloc}();\ \{x \mapsto \_\}$$

- To access a normal location, you must own it:

$$\{x \mapsto v\}\ t = *x;\ \{x \mapsto v \wedge t = v\}$$
$$\{x \mapsto v\}\ *x = v';\ \{x \mapsto v'\}$$

Reasoning about SC accesses

- Model SC accesses as non-atomic accesses inside a CCR.
- Use concurrent separation logic (CSL)

$$J \vdash \{P\}\ C\ \{Q\}$$

- Rule for SC-atomic reads:

$$\frac{emp \vdash \{J * P\}\ t = *x;\ \{J * Q\}}{J \vdash \{P\}\ t = x.load(sc);\ \{Q\}}$$

Rules for release/acquire accesses

Relaxed separation logic [OOPSLA'13]

Ownership transfer by rel-acq synchronizations.

- Atomic allocation $\rightsquigarrow$ pick loc. invariant $\mathcal{Q}$.

$$\left\{\mathcal{Q}(v)\right\}\ x = \mathsf{alloc}(v);\ \left\{\mathbf{W}_{\mathcal{Q}}(x) * \mathbf{R}_{\mathcal{Q}}(x)\right\}$$

- Release write $\rightsquigarrow$ give away permissions.

$$\left\{\mathcal{Q}(v) * \mathbf{W}_{\mathcal{Q}}(x)\right\}\ x.\mathsf{store}(v, \mathit{rel});\ \left\{\mathbf{W}_{\mathcal{Q}}(x)\right\}$$

- Acquire read $\rightsquigarrow$ gain permissions.

$$\left\{\mathbf{R}_{\mathcal{Q}}(x)\right\}\ t = x.\mathsf{load}(\mathit{acq});\ \left\{\mathcal{Q}(t) * \mathbf{R}_{\mathcal{Q}[t:=\mathsf{emp}]}(x)\right\}$$

Release-acquire synchronization: message passing

Initially $a = x = 0$. Let $J(v) \stackrel{\text{def}}{=} v = 0 \vee \&a \mapsto 5$.

$\{\&a \mapsto 0 * \mathbf{W}_J(x)\}$	$\{\mathbf{R}_J(x)\}$
$a = 5;$	**while** $(x.\text{load}(acq) == 0);$
$\{\&a \mapsto 5 * \mathbf{W}_J(x)\}$	$\{\&a \mapsto 5\}$
$x.\text{store}(release, 1);$	$\text{print}(a);$
$\{\mathbf{W}_J(x)\}$	$\{\&a \mapsto 5\}$

PL consequences:

Ownership transfer works!

Mutual exclusion locks

Let $\mathcal{Q}_J(v) \stackrel{\text{def}}{=} (v = 0 \wedge \text{emp}) \vee (v = 1 \wedge J)$

$Lock(x, J) \stackrel{\text{def}}{=} \mathbf{W}_{\mathcal{Q}_J}(x) * \mathbf{R}^{\text{CAS}}_{\mathcal{Q}_J}(x)$

$new\text{-}lock() \stackrel{\text{def}}{=}$

$\{J\}$

$res = \mathbf{alloc}(1)$

$\{Lock(res, J)\}$

$unlock(x) \stackrel{\text{def}}{=}$

$\{J * Lock(x, J)\}$

$x.\text{store}(1, rel)$

$\{Lock(x, J)\}$

$lock(x) \stackrel{\text{def}}{=}$

$\{Lock(x, J)\}$

repeat

$\{Lock(x, J)\}$

$y = x.\text{CAS}(1, 0, acq, rlx)$

$\left\{Lock(x, J) * \begin{pmatrix} y{=}0 \wedge \text{emp} \\ \vee\ y{=}1 \wedge J \end{pmatrix}\right\}$

until $y \neq 0$

$\{J * Lock(x, J)\}$

Slogan

Relaxed program logics
are good tools for
understanding
weak memory models