Debugging and improving the C/C++11 memory model

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

January 2016

The C11 memory model

Defines the semantics of
concurrent memory accesses in C/C++.

Standardised by ISO C/C++ 2011.

Used:

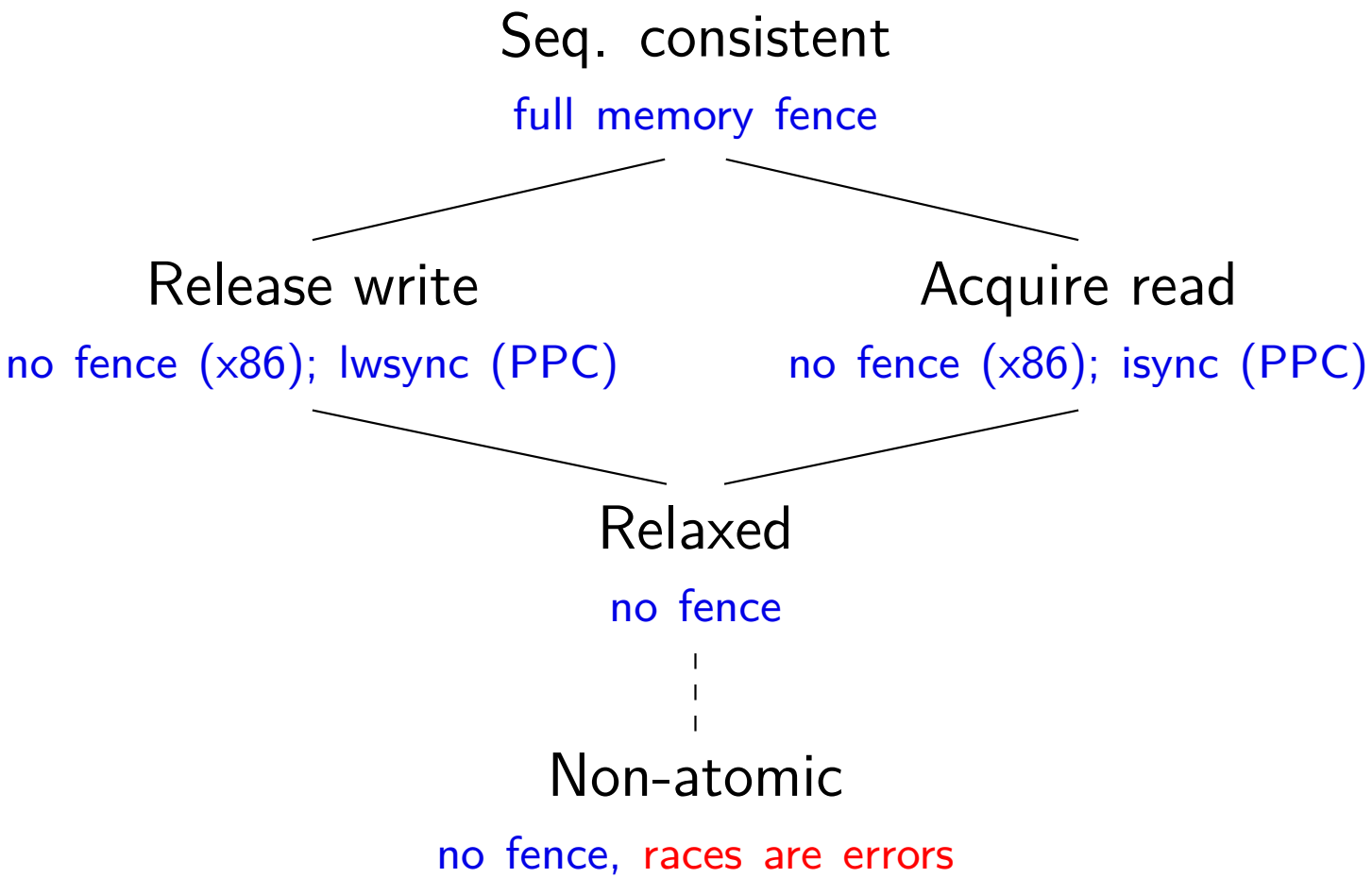
- By several POPL/PLDI/OOPSLA papers
- Internally by LLVM IR
- Indirectly by every program

The C11 memory model: Atomics

Two types of locations

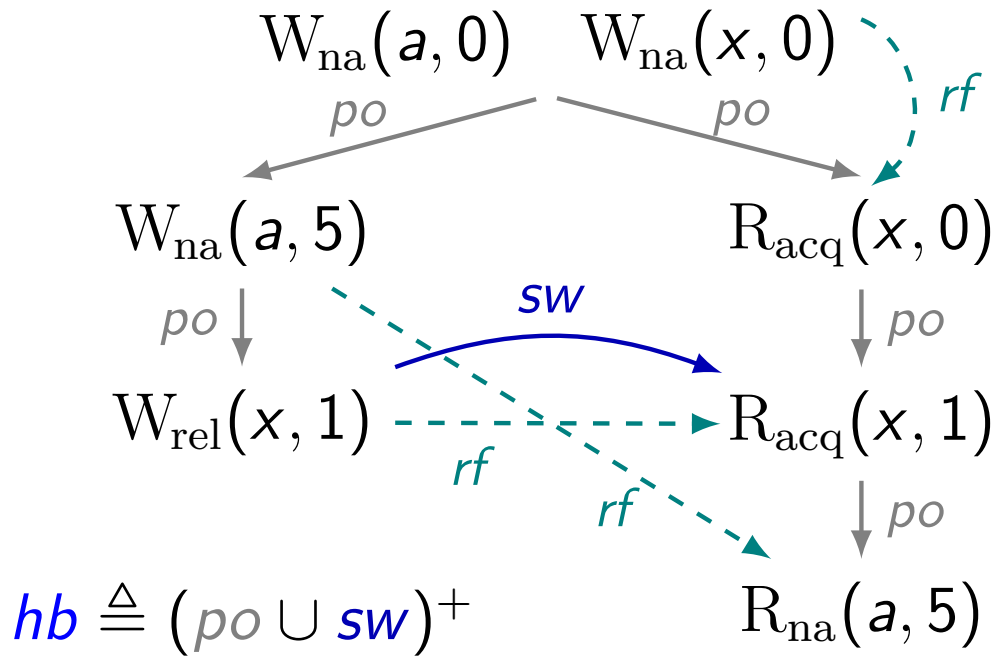
Ordinary (Non-Atomic)	**Atomic**
Races are errors	Welcome to the expert mode

The C11 memory model: a spectrum of accesses



Explicit primitives for fences

An execution in C11: actions and relations (and axioms)



Initially $a = x = 0$.

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

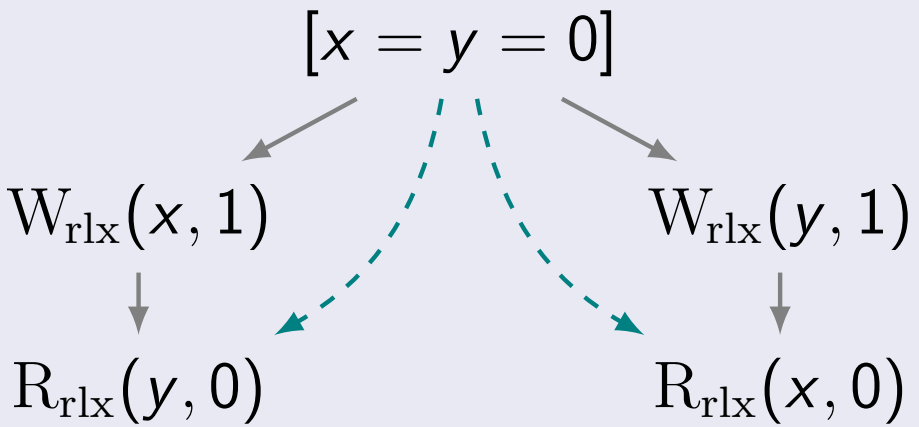
Relaxed behaviour: store buffering

Initially $x = y = 0$.

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

This can return $t_1 = t_2 = 0$.

Justification



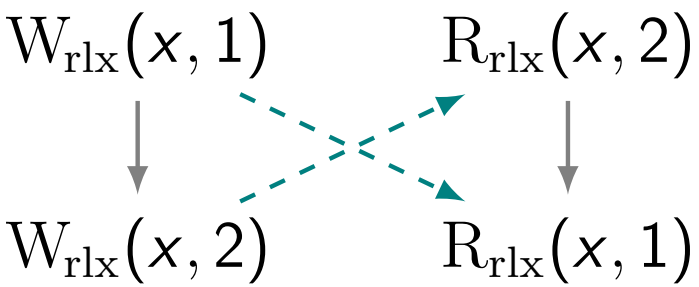
Behaviour observed on x86/Power/ARM

Coherence

Programs with a single shared variable behave as under SC.

$$\begin{array}{l} x.\mathsf{store}(1, \mathit{rlx}); \\ x.\mathsf{store}(2, \mathit{rlx}); \end{array} \;\Big\|\; \begin{array}{l} a = x.\mathsf{load}(\mathit{rlx}); \\ b = x.\mathsf{load}(\mathit{rlx}); \end{array}$$

The outcome $a = 2 \wedge b = 1$ is forbidden.

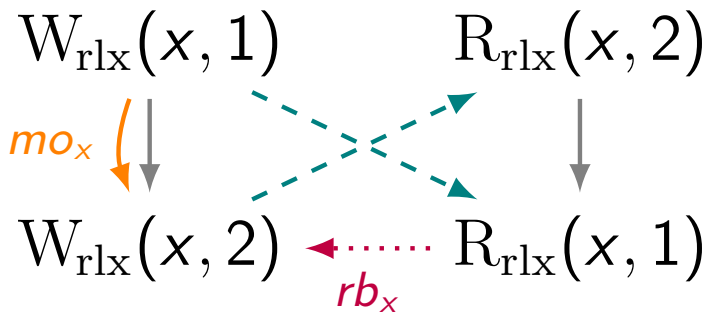


Coherence

Programs with a single shared variable behave as under SC.

$$\begin{array}{l} x.\mathrm{store}(1, rlx); \\ x.\mathrm{store}(2, rlx); \end{array} \quad \Big\| \quad \begin{array}{l} a = x.\mathrm{load}(rlx); \\ b = x.\mathrm{load}(rlx); \end{array}$$

The outcome $a = 2 \wedge b = 1$ is forbidden.



- Modification order, mo_x, total order of writes to x.
- Reads-before : $rb_x \triangleq (rf^{-1}; mo_x) \cap (\neq)$
- Coherence : $hb \cup rf_x \cup mo_x \cup rb_x$ is acyclic for all x.

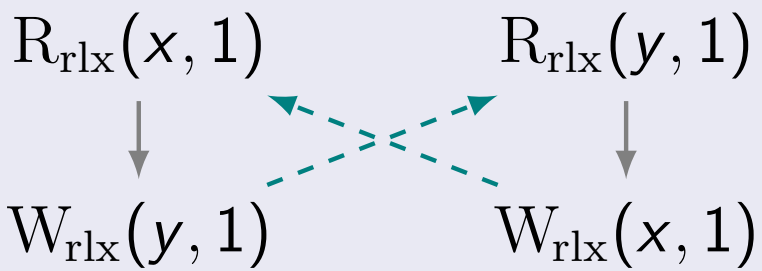
Causality cycles with relaxed accesses

Initially $x = y = 0$.

$$\textbf{if } (x.load(rlx) == 1) \quad \| \quad \textbf{if } (y.load(rlx) == 1)$$
$$\quad y.store(1, rlx); \quad \| \quad \quad x.store(1, rlx);$$

C11 allows the outcome $x = y = 1$.

Justification



Relaxed accesses don't synchronize

No causality cycles with non-atomics

Initially $x = y = 0$.

$$\begin{array}{r} \textbf{if } (x == 1) \\ y = 1; \end{array} \quad \Big\| \quad \begin{array}{r} \textbf{if } (y == 1) \\ x = 1; \end{array}$$

C11 forbids the outcome $x = y = 1$.

Justification

Non-atomic read axiom:

$$rf \cap (\_ \times NA) \subseteq hb$$

Is the C11 memory model definition...

1. Mathematically sane?
   - *For example, it is monotone.*

2. Not too weak?
   - *Provides useful reasoning principles.*

3. Not too strong?
   - *Can be implemented efficiently.*

4. Actually useful?
   - *Admits the intended program optimisations.*

Is the C11 memory model definition...

1. Mathematically sane?
   - *For example, it is monotone.*

2. Not too weak?
   - *Provides useful reasoning principles.*

3. Not too strong?

   ✓ Compilation to x86/Power/ARM.

4. Actually useful?
   - *Admits the intended program optimisations.*

Is the C11 memory model definition...

1. Mathematically sane?
   - *For example, it is monotone.*

2. Not too weak?

   ≈ Reasoning principles for C11 subsets.

3. Not too strong?

   ✓ Compilation to x86/Power/ARM.

4. Actually useful?
   - *Admits the intended program optimisations.*

Is the C11 memory model definition...

1. Mathematically sane?

   ✗ No, it is not monotone.

2. Not too weak?

   ≈ Reasoning principles for C11 subsets.

3. Not too strong?

   ✓ Compilation to x86/Power/ARM.

4. Actually useful?
   - *Admits the intended program optimisations.*

Is the C11 memory model definition...

1. Mathematically sane?

   ✗ No, it is not monotone.

2. Not too weak?

   ≈ Reasoning principles for C11 subsets.

3. Not too strong?

   ✓ Compilation to x86/Power/ARM.

4. Actually useful?

   ✗ No, it disallows intended program transformations.

Is the C11 memory model definition...

1. Mathematically sane?

   ✗ No, it is not monotone.

2. Not too weak?

   ≈ Reasoning principles for C11 subsets.

3. Not too strong?

   ≈ Compilation to x86/Power/ARM.

4. Actually useful?

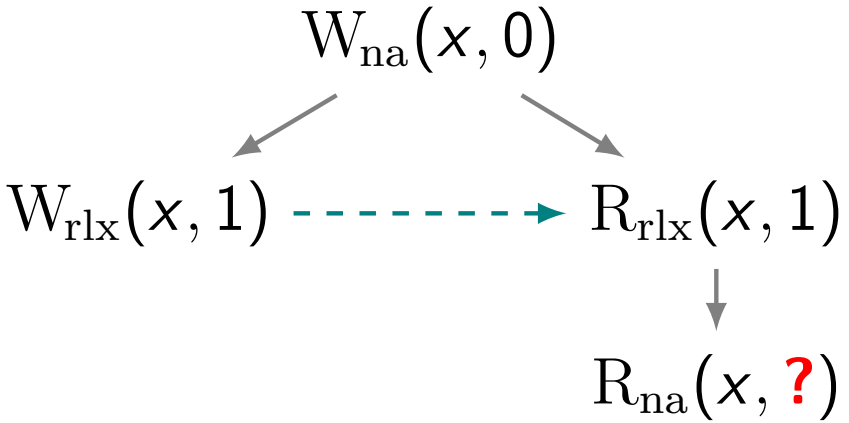
   ✗ No, it disallows intended program transformations.

Non-atomic reads of atomic variables are unsound!

Initially, $x = 0$.

$$x.\mathsf{store}(1, \mathit{rlx}); \;\Big\|\; \begin{array}{l}\textbf{if } (x.\mathsf{load}(\mathit{rlx}) == 1)\\ \quad t = (\mathsf{int})\, x;\end{array}$$

The program can get stuck!



- Reading 0 contradicts coherence.
- Reading 1 contradicts the non-atomic read axiom.

Sequentialisation is invalid

Initially, $a = x = y = 0$.

$$a = 1; \;\Big\|\; \textbf{if } (x.\text{load}(rlx) == 1) \;\; \textbf{if } (a == 1) \;\; y.\text{store}(1, rlx); \;\Big\|\; \textbf{if } (y.\text{load}(rlx) == 1) \;\; x.\text{store}(1, rlx);$$

The only possible output is:

$$a = 1, \quad x = y = 0\,.$$

Recall the non-atomic read axiom:

$$rf \cap (\_ \times NA) \subseteq hb$$

Tentative fixes

Remove non-atomic read axiom.

- gives extremely weak guarantees, if any

In addition, forbid $(hb \cup rf)$-cycles.

- rules out causal loops
- forbids some reorderings
- more costly on ARM/Power

Or alternatively forbid $(hb \cup rf)$-cycles with NA accesses.

- allows more racy behaviours
- forbids some reorderings

Tentative fixes

Remove non-atomic read axiom.

- gives extremely weak gua… …y

In addition, forbid (hb …

- rules out caus…
- forbids som… …s
- more costly o… …M/Power

Open problem

Or alternatively forbid ($hb \cup rf$)-cycles with NA accesses.

- allows more racy behaviours
- forbids some reorderings

Monotonicity

"Adding synchronisation should not introduce new behaviours"

Examples:

- Reducing parallelism, $C_1 \| C_2 \rightsquigarrow C_1 \,;\, C_2$
- Expression evaluation linearisation:

$$x = a + b\,; \quad \rightsquigarrow \quad t_1 = a\,;\, t_2 = b\,;\, x = t_1 + t_2\,;$$

- Adding a memory fence
- Strengthening the access mode of an operation
- (Roach motel reorderings)

Other problems fixed (POPL'15, POPL'16)

The axiom of SC reads is too weak.

- Makes strengthening unsound.

The axioms of SC fences are too weak.

- They do not guarantee sequential consistency.

The definition of release sequences is too strong.

- Removing $(po \cup rf)$-final events is unsound.

Transformation correctness

Valid instruction reorderings $a\,;b \rightsquigarrow b\,;a$ (POPL'15)

$\downarrow a \setminus b \rightarrow$	$\mathsf{R}_{\neq\mathrm{sc}}$	R_{sc}	W_{na}	$\mathsf{W}_{\mathrm{rlx}}$	$\mathsf{W}_{\sqsupseteq\mathrm{rel}}$	$\mathsf{C}_{\mathrm{rlx}\mid\mathrm{acq}}$	$\mathsf{C}_{\sqsupseteq\mathrm{rel}}$	$\mathsf{F}_{\mathrm{acq}}$	$\mathsf{F}_{\mathrm{rel}}$
R_{na}	✓	✓	✓	✓	✗	✓	✗	✓	✗
$\mathsf{R}_{\mathrm{rlx}}$	✓	✓	✓	(✓)	✗	(✓)	✗	✗	✗
$\mathsf{R}_{\sqsupseteq\mathrm{acq}}$	✗	✗	✗	✗	✗	✗	✗	✓	✗
$\mathsf{W}_{\neq\mathrm{sc}}$	✓	✓	✓	✓	✗	✓	✗	✓	✗
W_{sc}	✓	✗	✓	✓	✗	✓	✗	✓	✗
$\mathsf{C}_{\mathrm{rlx}\mid\mathrm{rel}}$	✓	✓	✓	(✓)	✗	(✓)	✗	✗	✗
$\mathsf{C}_{\sqsupseteq\mathrm{acq}}$	✗	✗	✗	✗	✗	✗	✗	✓	✗
$\mathsf{F}_{\mathrm{acq}}$	✗	✗	✗	✗	✗	✗	✗	=	✗
$\mathsf{F}_{\mathrm{rel}}$	✓	✓	✓	✗	✓	✗	✓	✓	=

Redundant instruction eliminations (POPL'15)

Overwritten write:

$$\begin{array}{rll} & x.\mathsf{store}(v, M)\,;\, C\,;\, x.\mathsf{store}(v', M) & C \text{ has no rel} \\ \rightsquigarrow & C\,;\, x.\mathsf{store}(v', M) & \& \text{ no } x \text{ accesses} \end{array}$$

Read after write:

$$\begin{array}{rll} & x.\mathsf{store}(v, M)\,;\, C\,;\, t = x.\mathsf{load}(M') & C \text{ has no acq} \\ \rightsquigarrow & x.\mathsf{store}(v, M)\,;\, C\,;\, t = v & \& \text{ no } x \text{ accesses} \end{array}$$

Read after read:

$$\begin{array}{rll} & t = x.\mathsf{load}(M)\,;\, C\,;\, t' = x.\mathsf{load}(M) & C \text{ has no acq} \\ \rightsquigarrow & t = x.\mathsf{load}(M)\,;\, C\,;\, t' = t & \& \text{ no } x \text{ accesses} \end{array}$$

Is DRF semantics really what we want?

Should these transformations be allowed?

1. **CSE over a lock acquire:**

$$\begin{array}{l} t_1 = X; \\ lock(); \\ t_2 = X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t_1 = X; \\ lock(); \\ t_2 = t_1; \end{array}$$

   If X changes in between, the program is racy.

2. **Load hoisting:**

$$\begin{array}{l} \text{if}(c) \\ \quad r = X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t = X; \\ r = c \ ? \ t : r; \end{array}$$

   This may introduce a race, but the racy value is not used.

Allowing both is clearly wrong!

Consider the transformation sequence:

$$\begin{array}{l} \text{if } (c) \\ \quad r_1 = X; \\ lock(); \\ r_2 = X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t = X; \\ r_1 = c \ ? \ t : r_1; \\ lock(); \\ r_2 = X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t = X; \\ r_1 = c \ ? \ t : r_1; \\ lock(); \\ r_2 = t; \end{array}$$

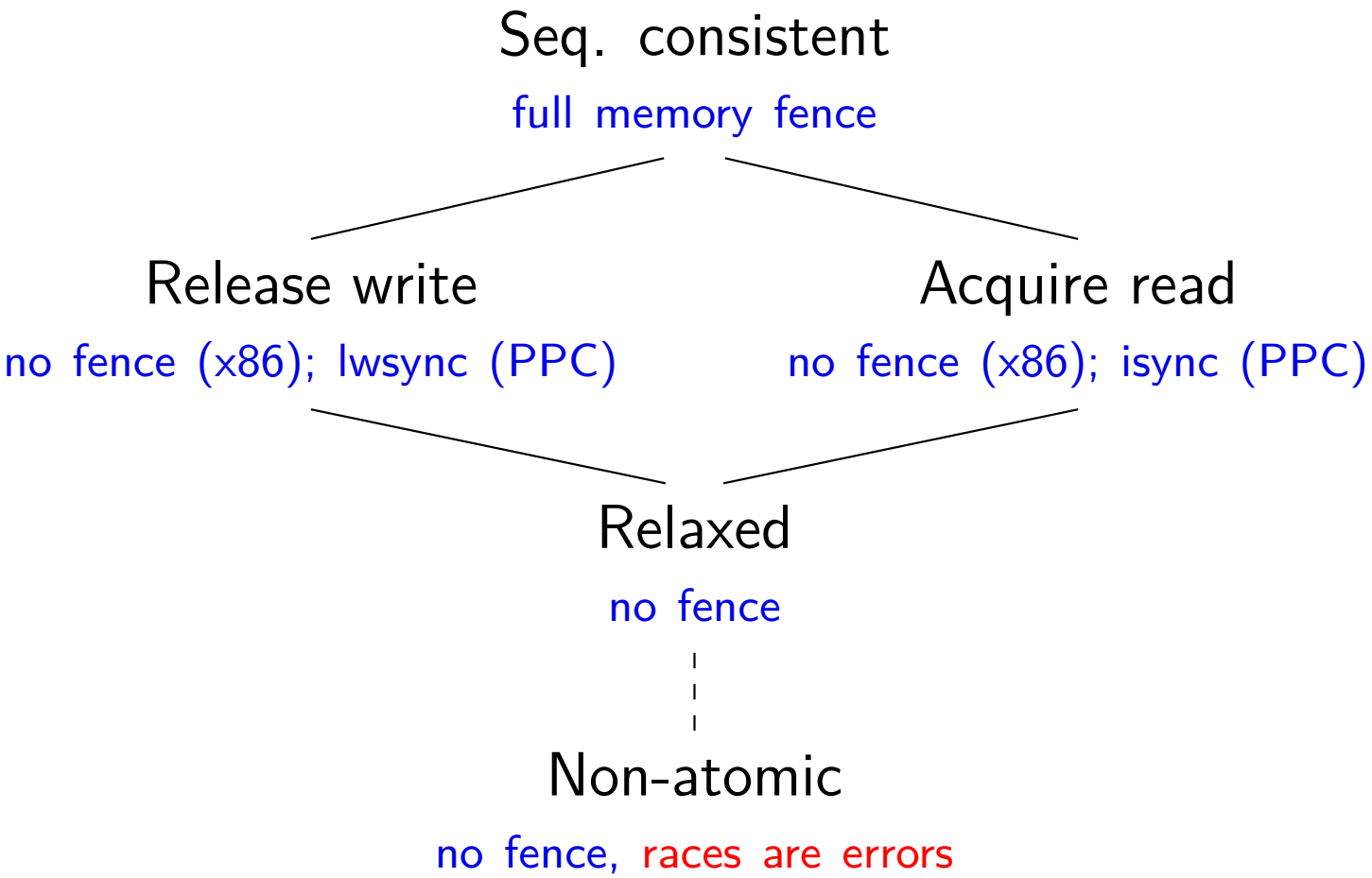
When c is false, X is moved out of the critical region!

So we have to forbid one transfomation.

- C11 forbids load hoisting, allows CSE over lock().
- LLVM allows load hoisting, forbids CSE over lock().

Taming the release-acquire fragment

Recall the spectrum of C11 access types



C11's release-acquire memory model

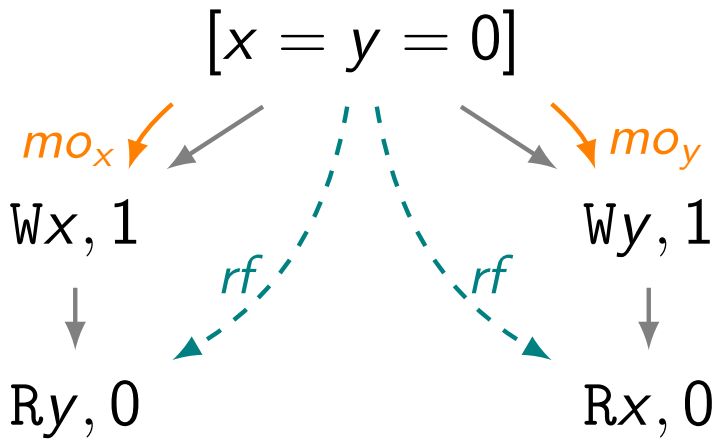
C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release

Store buffering

$$x = y = 0$$

$x := 1;$	$\parallel$	$y := 1;$
$\texttt{print}\ y$	$\parallel$	$\texttt{print}\ x$

both threads may print 0

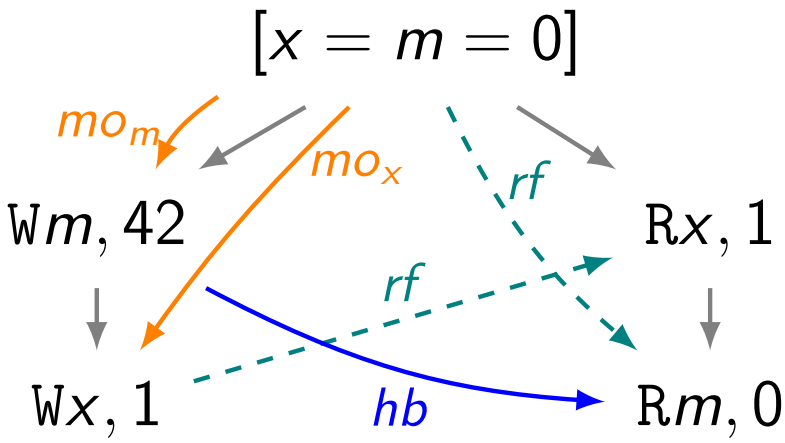


Message passing

$$x = m = 0$$

$m := 42;$	$\parallel$	$\texttt{while}\ x = 0$
$x := 1$	$\parallel$	$\texttt{skip};$
	$\parallel$	$\texttt{print}\ m$

only 42 may be printed



Good news

- Verified compilation schemes:
  - x86-TSO (trivial compilation) [Batty el al. '11]
  - Power [Batty el al. '12] [Sarkar el al. '12]
- RA supports intended optimizations:
  - In particular, write-read reordering (unlike SC):

$$\mathtt{W}x \rightarrow \mathtt{R}y \quad \rightsquigarrow \quad \mathtt{R}y \rightarrow \mathtt{W}x$$

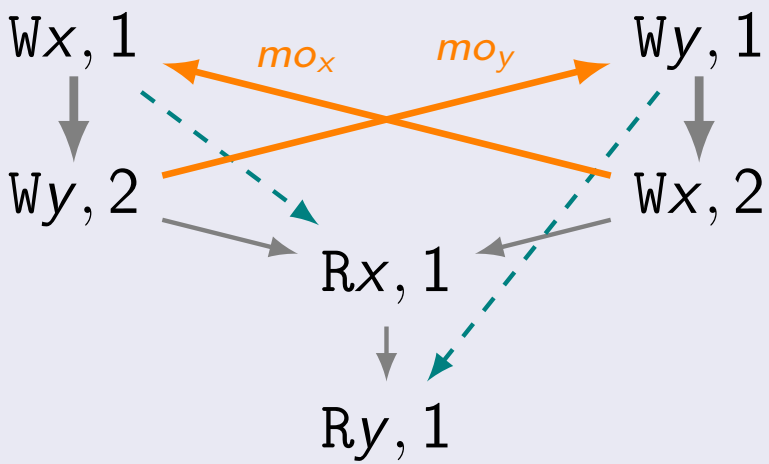
- DRF theorem:
  - No data races under SC ensures no weak behaviors
- Monotonicity:
  - Adding synchronization does not introduce new behaviors
- Program logics:
  - RSL [Vafeiadis and Narayan '13]
  - GPS [Turon et al. '14]
  - OGRA [Lahav and Vafeiadis '15]

Bad news: RA allows some unobersable behaviors

The following program may print 1, 1.

$$
\begin{array}{l||l}
x := 1; & y := 1; \\
y := 2; & x := 2; \\
\multicolumn{2}{c}{\texttt{print}\ x;} \\
\multicolumn{2}{c}{\texttt{print}\ y;}
\end{array}
$$

Justification



Strong release/acquire consistency

Definition (SRA-consistency)

An execution is *SRA-consistent* if it is RA-consistent and $hb \cup \bigcup_x mo_x$ is acyclic.

Proposition

If there are no write-write races, then SRA and RA coincide.

Same product, better price:

- Same compiler optimizations are sound.
- Compilation to x86-TSO and Power is still correct.
- No better deal for Power:

Power model restricted to RA accesses = SRA

(based on Power's declarative model of [Alglave et al. '14])

Example: the SRA machine (first attempt)

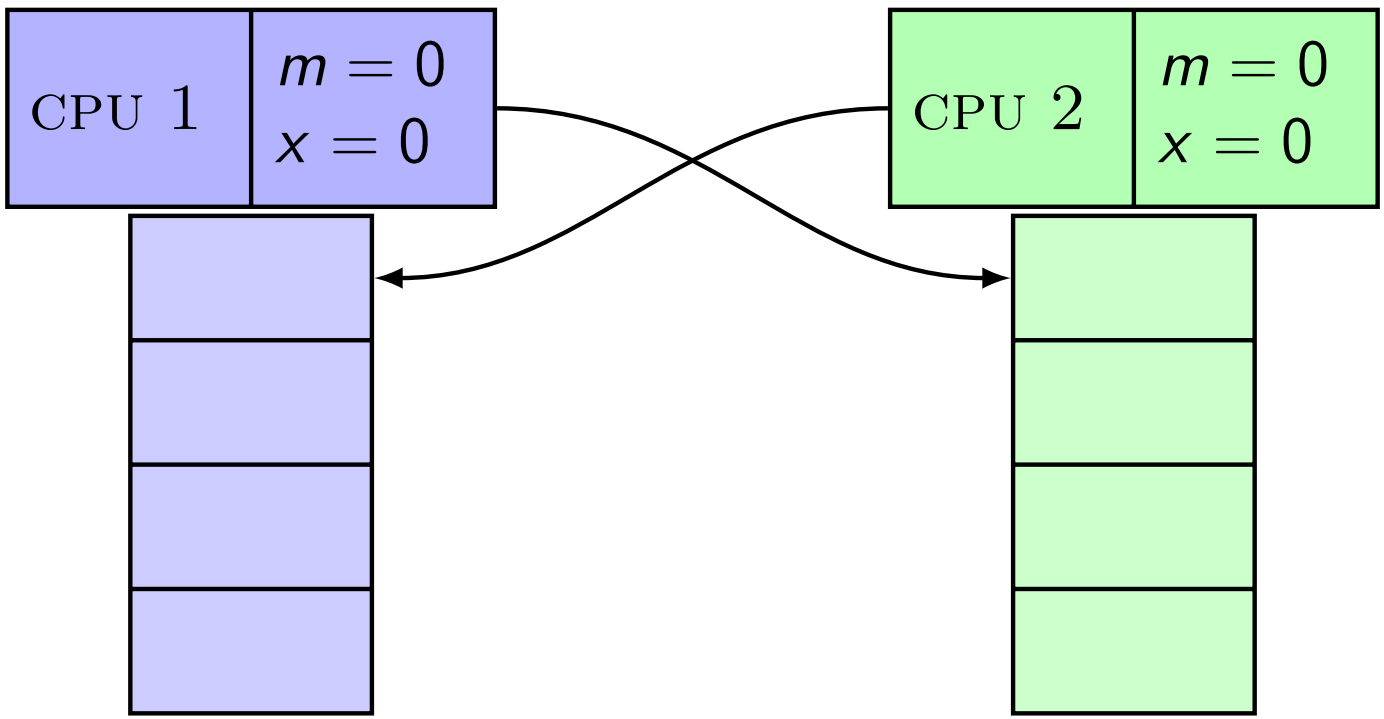
Message passing

$$m = x = 0$$

- $m := 42$;
  $x := 1$

$\parallel$

- `while` $x = 0$
  `skip`;
  `print` m

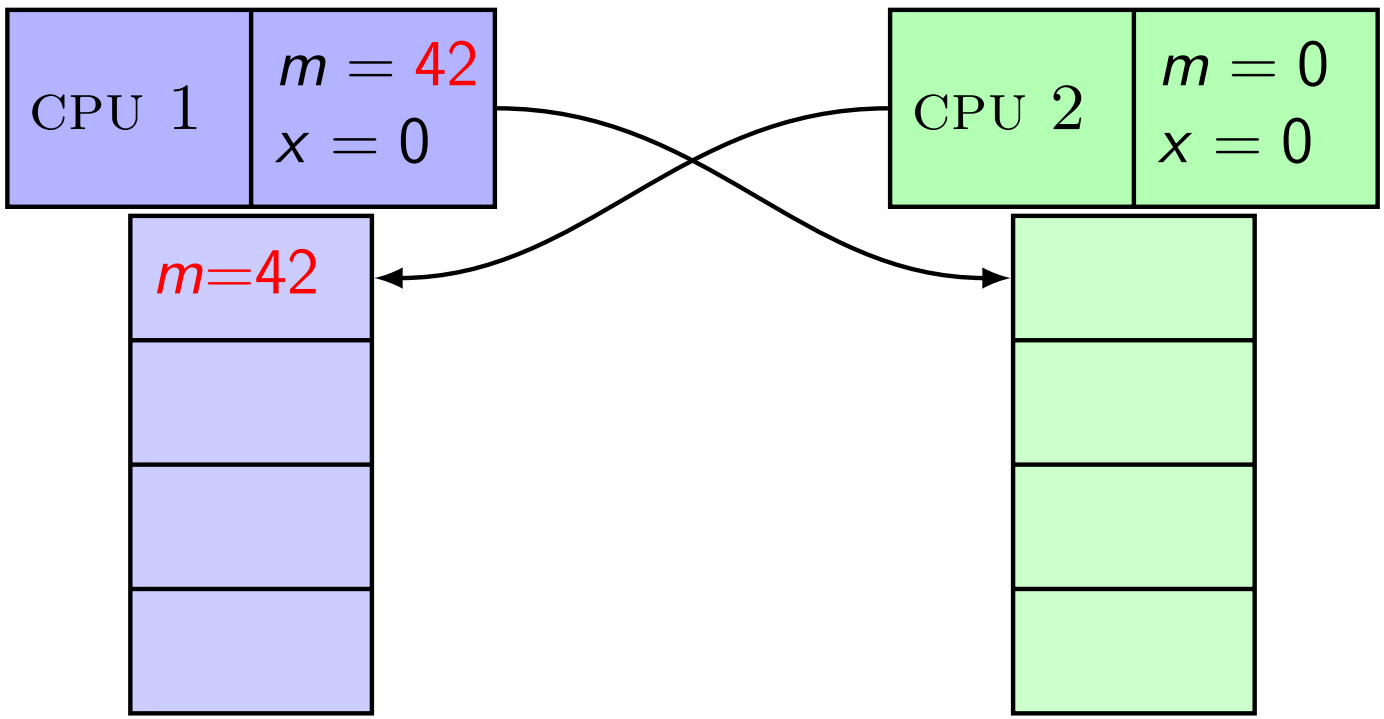


Example: the SRA machine (first attempt)

Message passing

$$m = x = 0$$

$$\begin{array}{l} m := 42; \\ \blacktriangleright\ x := 1 \end{array} \quad \Big\| \quad \begin{array}{l} \blacktriangleright\ \texttt{while}\ x = 0 \\ \qquad \texttt{skip}; \\ \texttt{print}\ m \end{array}$$

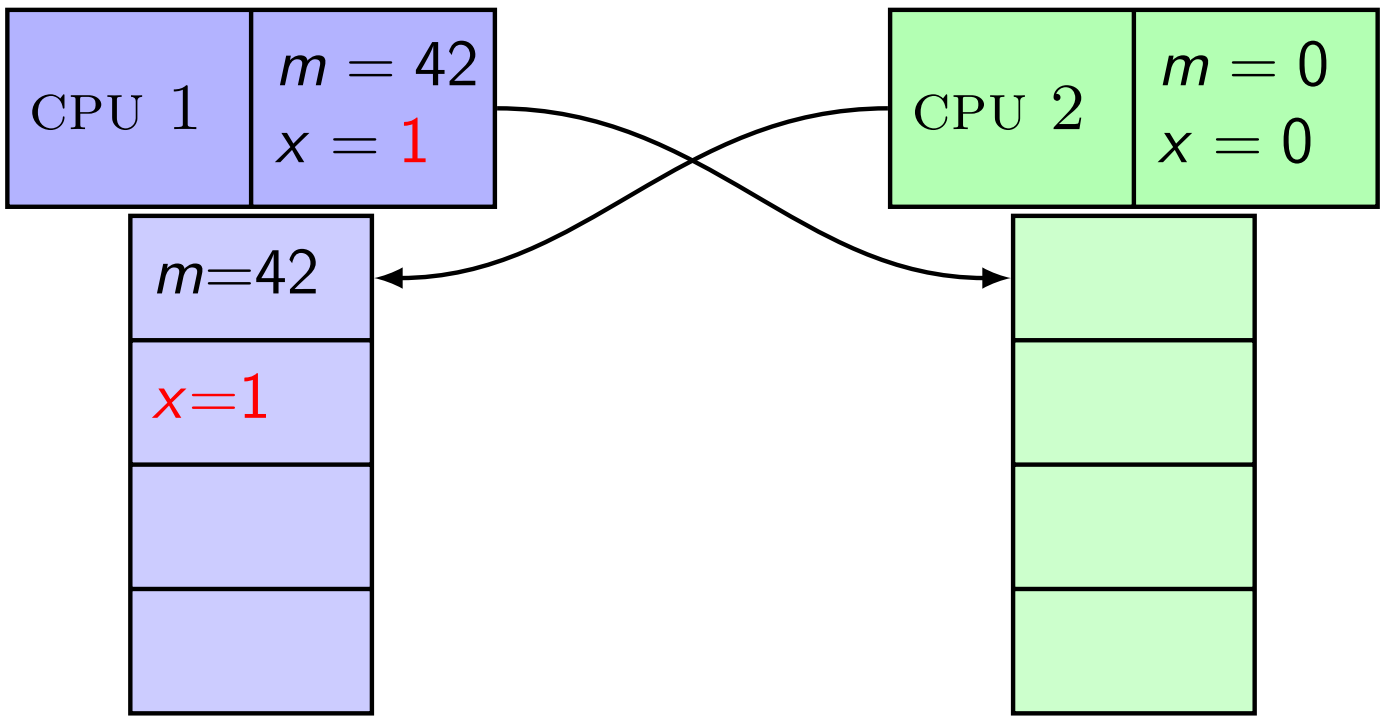


Example: the SRA machine (first attempt)

Message passing

$$m = x = 0$$

$$\begin{array}{l} m := 42; \\ x := 1 \end{array} \quad \Big\| \quad \begin{array}{l} \blacktriangleright \texttt{while } x = 0 \\ \qquad \texttt{skip}; \\ \texttt{print } m \end{array}$$

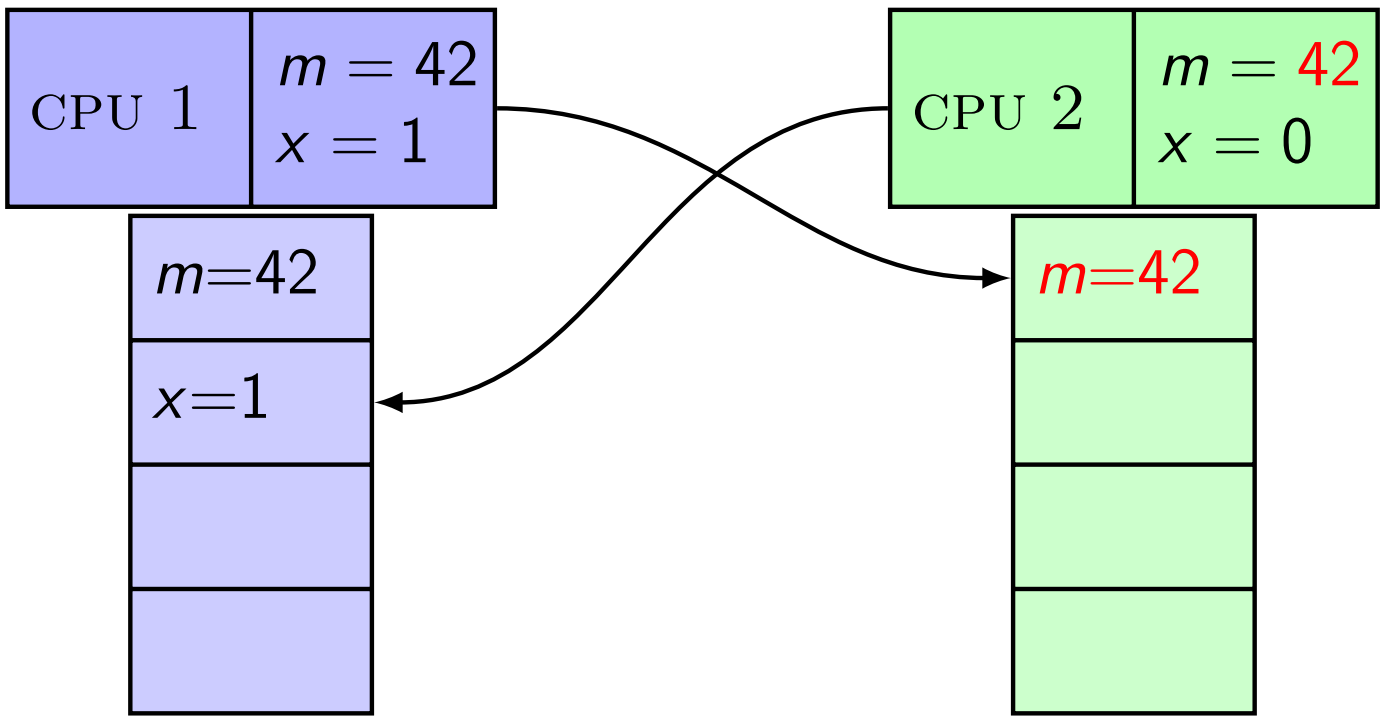


Example: the SRA machine (first attempt)

Message passing

$$m = x = 0$$

$$m := 42; \quad x := 1 \quad \Big\| \quad \blacktriangleright \texttt{while } x = 0 \texttt{ skip; print } m$$

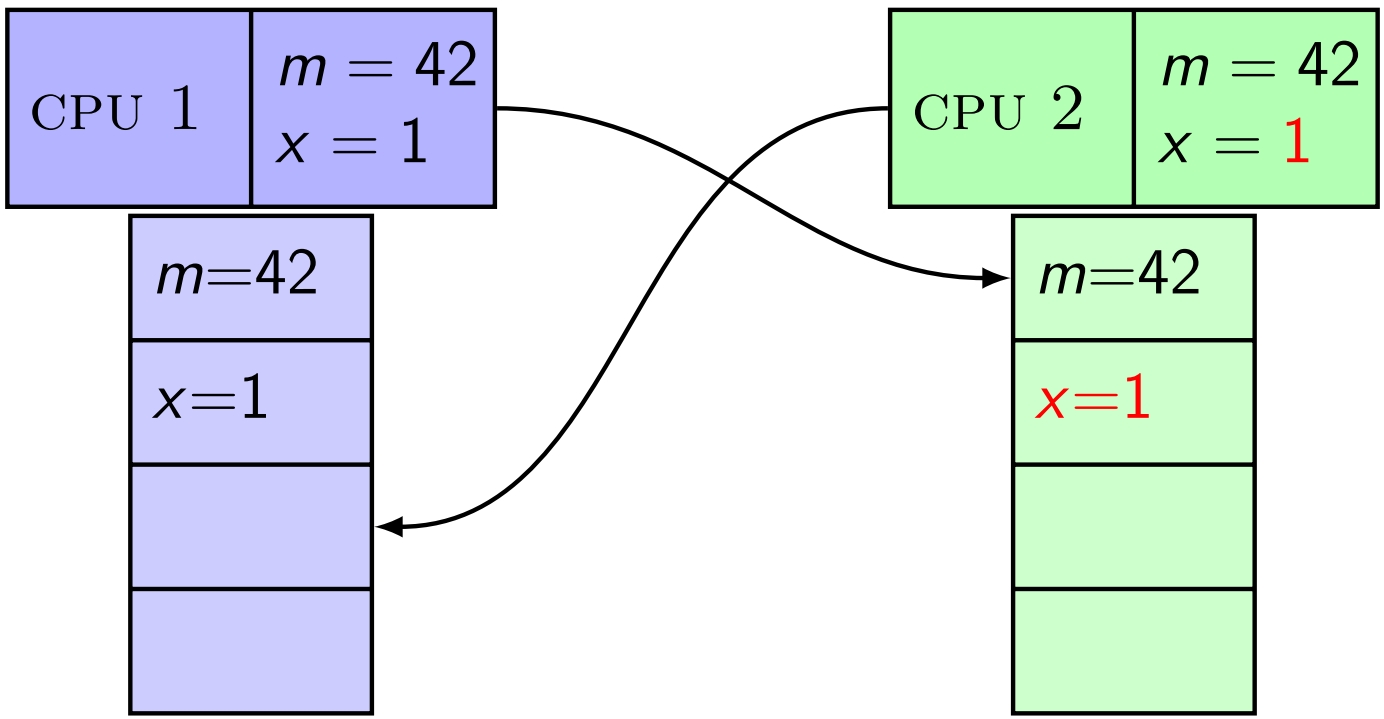


Example: the SRA machine (first attempt)

Message passing

$$m = x = 0$$

$$\begin{array}{l} m := 42; \\ x := 1 \end{array} \quad \Big\| \quad \begin{array}{l} \blacktriangleright \texttt{while } x = 0 \\ \qquad \texttt{skip;} \\ \texttt{print } m \end{array}$$

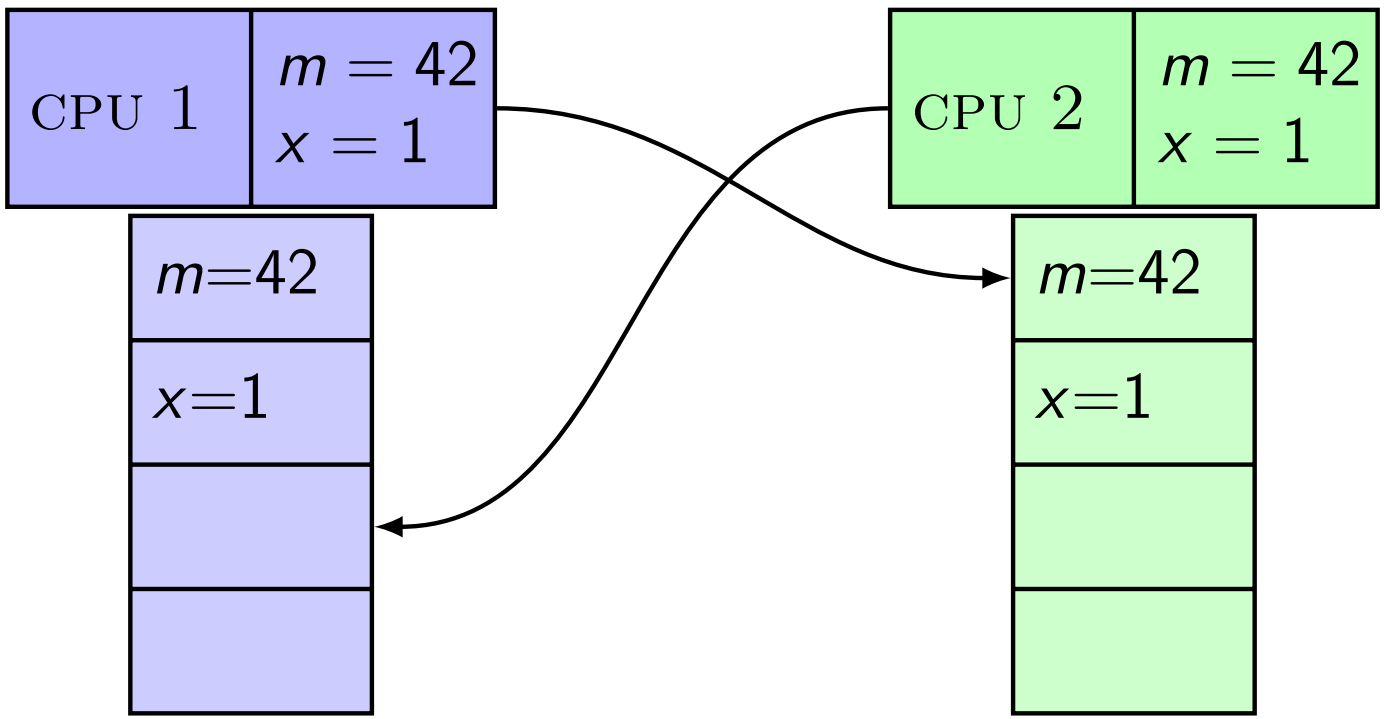


Example: the SRA machine (first attempt)

Message passing

$$m = x = 0$$

$m := 42;$	$\parallel$	`while` $x = 0$
$x := 1$		`skip;`
		▶ `print` m



Timestamps

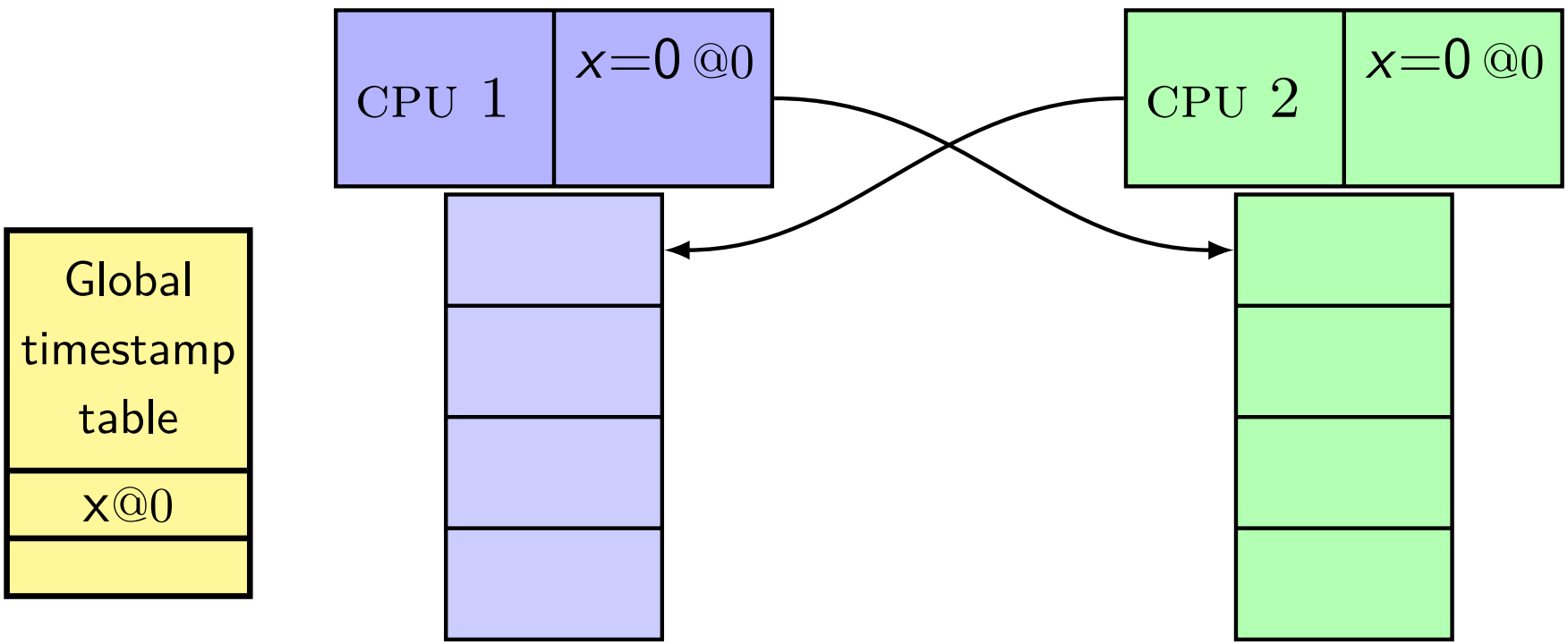
$$\begin{array}{l} x := 1; \\ \texttt{print}\ x \end{array} \quad \Big\| \quad \begin{array}{l} x := 2; \\ \texttt{print}\ x \end{array}$$

If the first thread prints 2, the second thread cannot print 1.

Timestamps

▶ $x := 1$; `print` x	▶ $x := 2$; `print` x

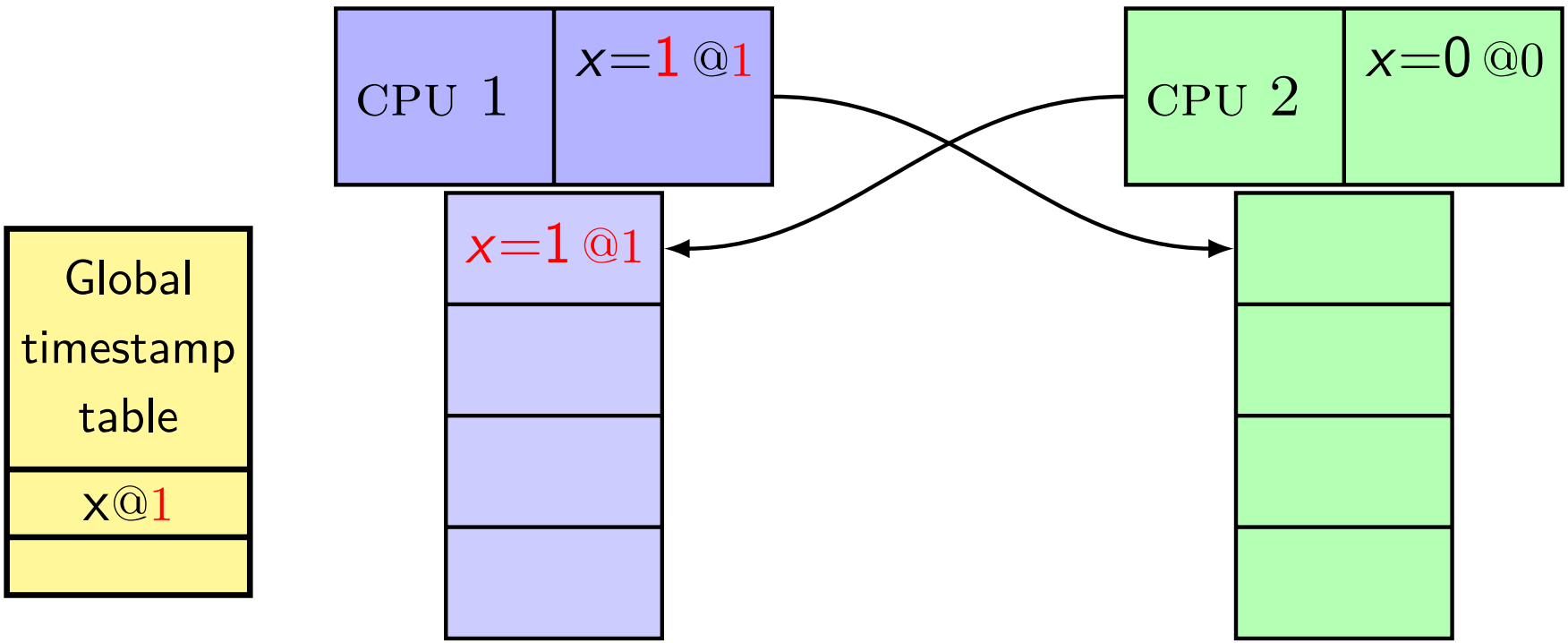
If the first thread prints 2, the second thread cannot print 1.



Timestamps

$$\begin{array}{l} x := 1; \\ \blacktriangleright\ \texttt{print}\ x \end{array} \quad \Big\| \quad \begin{array}{l} \blacktriangleright\ x := 2; \\ \texttt{print}\ x \end{array}$$

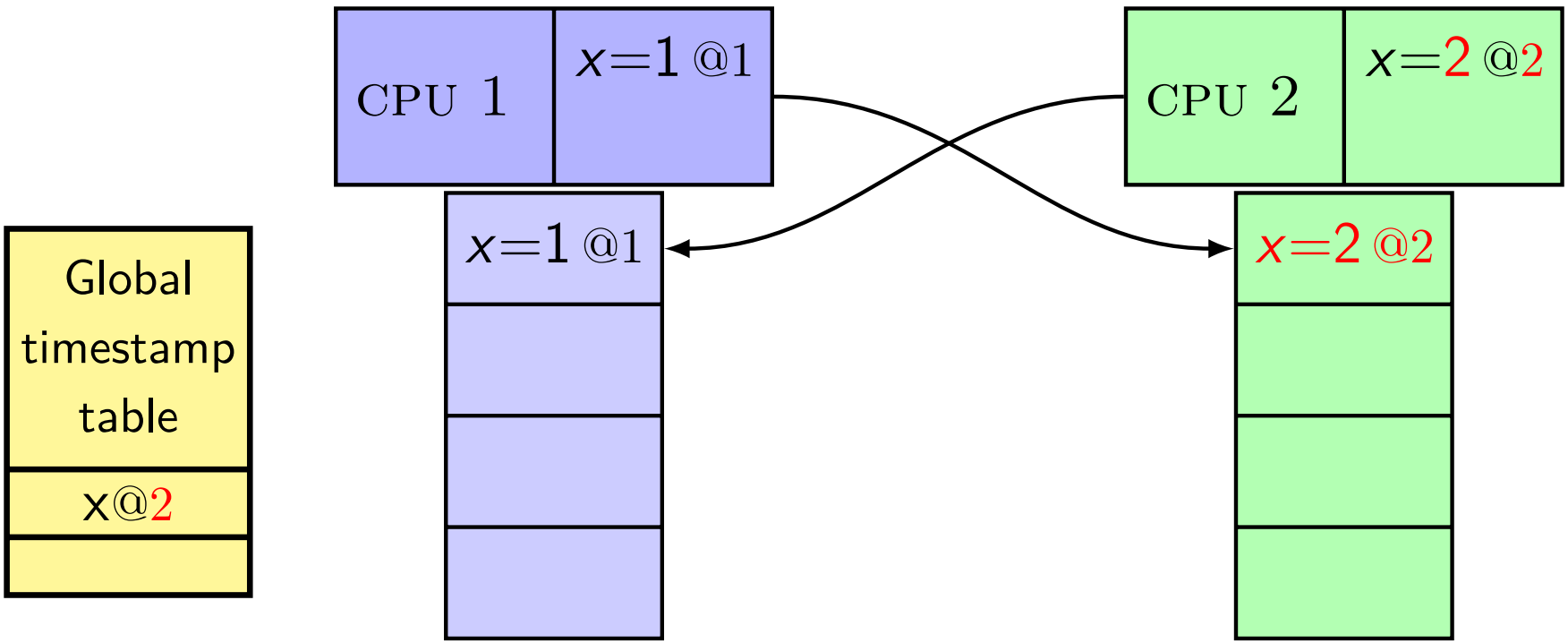
If the first thread prints 2, the second thread cannot print 1.



Timestamps

$x := 1;$ ▶ `print` x $\parallel$ $x := 2;$ ▶ `print` x

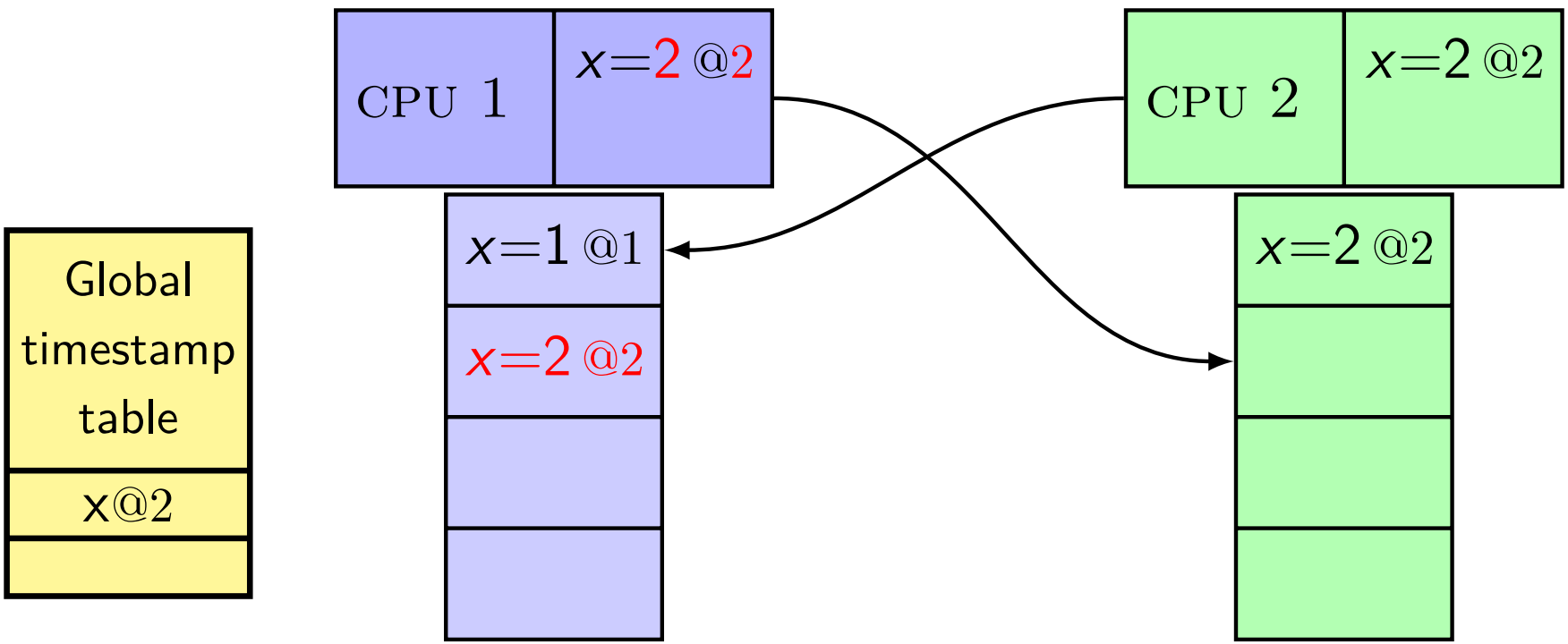
If the first thread prints 2, the second thread cannot print 1.



Timestamps

$x := 1;$ ▶ `print` x	$x := 2;$ ▶ `print` x

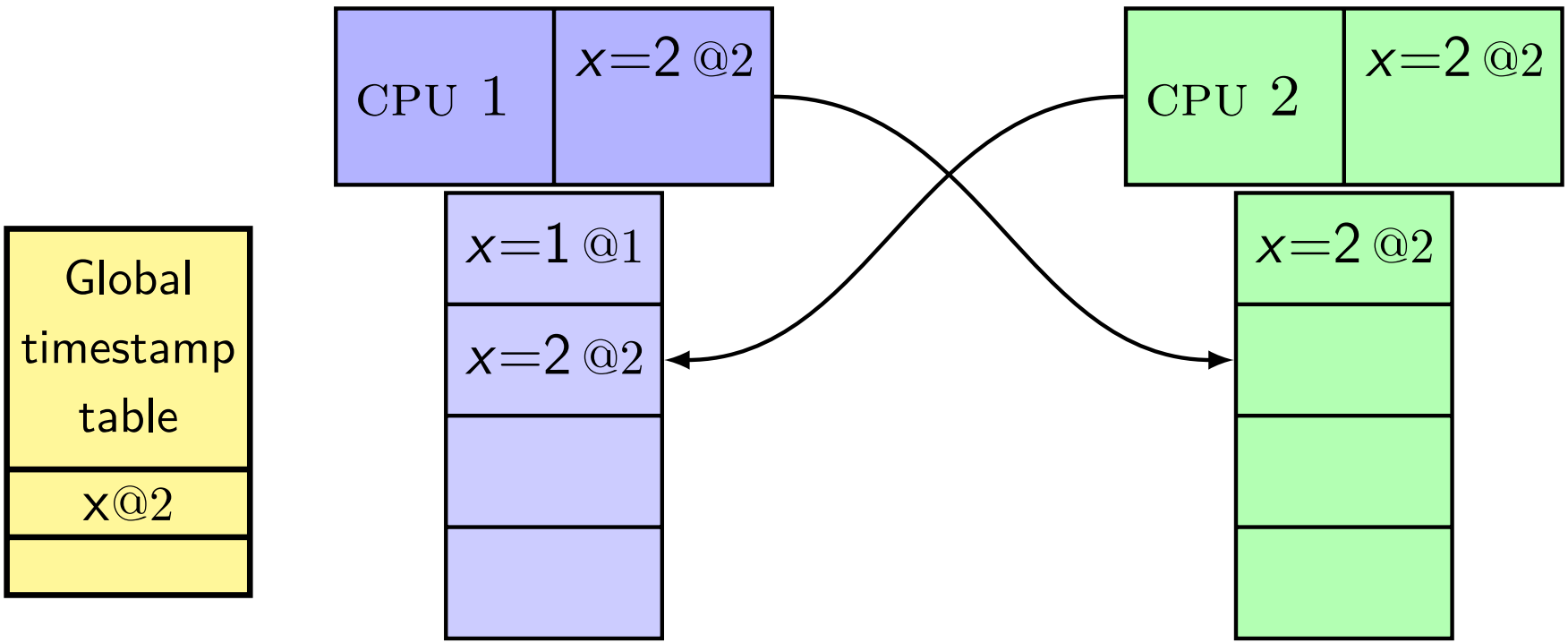
If the first thread prints 2, the second thread cannot print 1.



Timestamps

$x := 1;$ ▶ `print` x ‖ $x := 2;$ ▶ `print` x

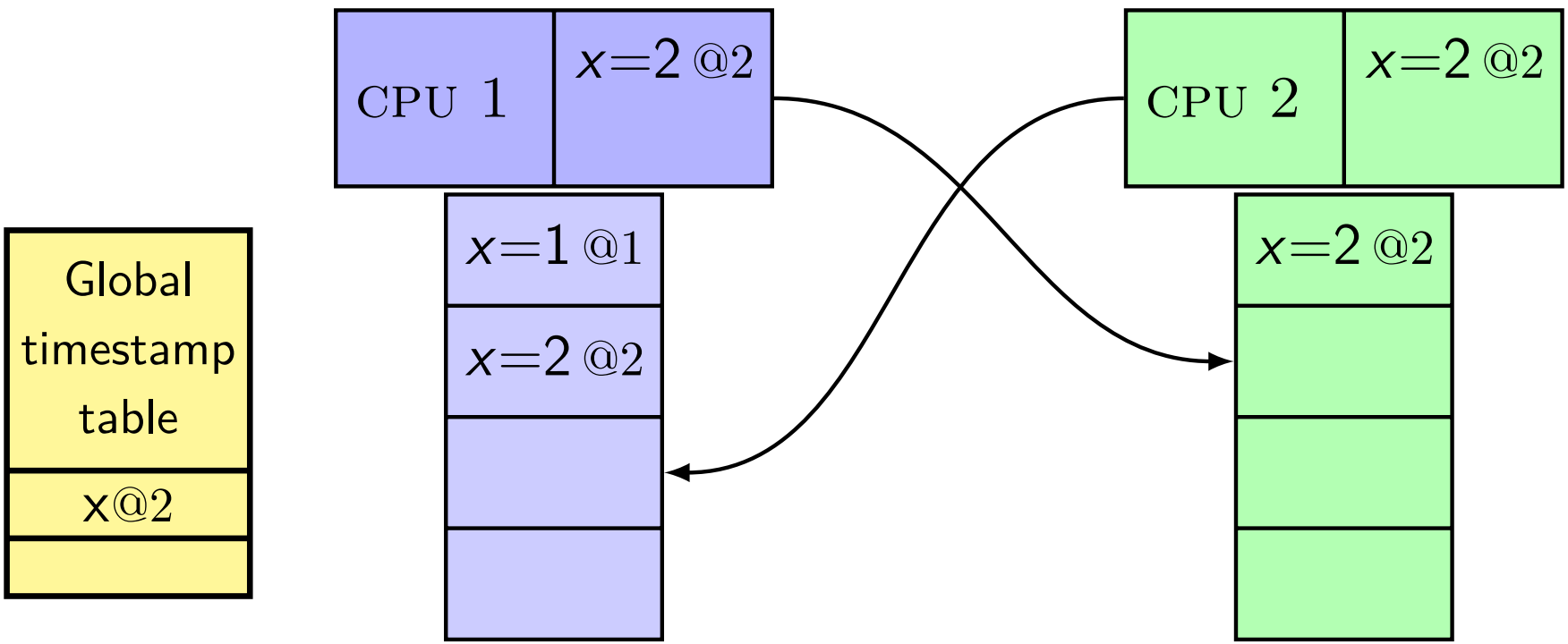
If the first thread prints 2, the second thread cannot print 1.



Timestamps

$x := 1;$ ▶ `print` x $\quad\|\quad$ $x := 2;$ ▶ `print` x

If the first thread prints 2, the second thread cannot print 1.



Summary

The C11 memory model is broken.

- But largely fixable.
- Ruling out causality cycles is still open.
- The "catch-fire" semantics is not ideal for compilers.

The release/acquire fragment of C11:

- Strikes good balance between performance and programmability.
- With no additional cost, can be strengthened to:
  - forbid unobservable weak behaviors and
  - admit intuitive operational semantics.