Weak memory consistency

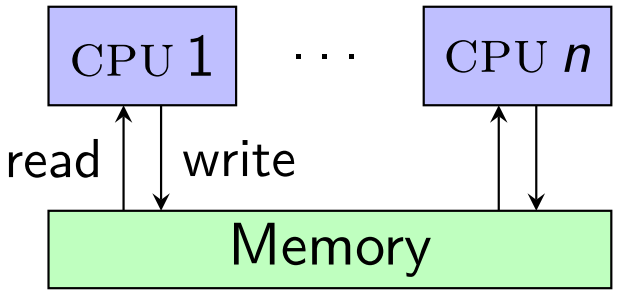
Viktor Vafeiadis
MPI-SWS

University of Kaiserslautern, December 2019

The illusion of sequential consistency

Sequential consistency (SC)

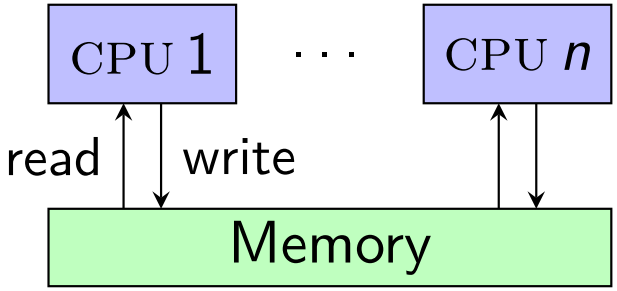
- The standard simplistic concurrency model.
- Threads access shared memory in an interleaved fashion.



The illusion of sequential consistency

Sequential consistency (SC)

- The standard simplistic concurrency model.
- Threads access shared memory in an interleaved fashion.



But...

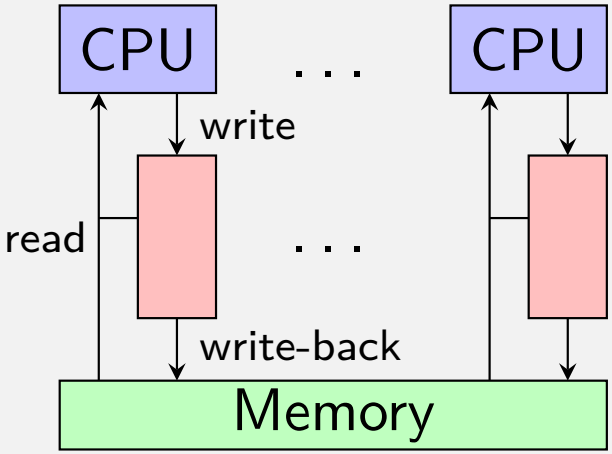
- No multicore processor implements SC.
- Compiler optimizations invalidate SC.

Weak consistency

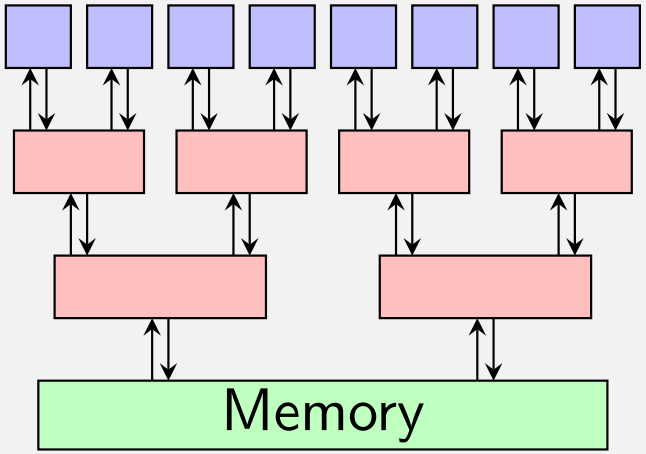
Hardware provides weak consistency.

- Weak memory models $\rightsquigarrow$ semantics of shared memory.
- Every hardware architecture has its own WMM: x86-TSO, ARM, Power, Itanium.

x86-TSO model (2010)



ARMv8 model (2016)

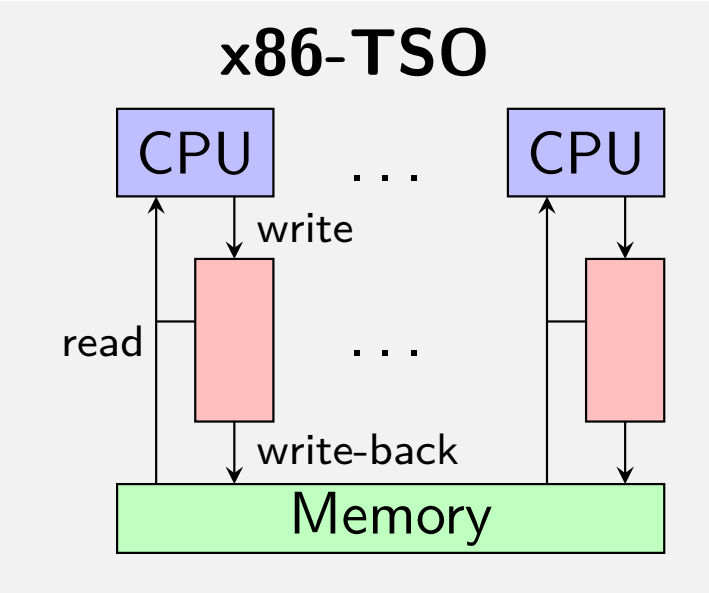


Weak consistency examples

Store buffering (SB)

Initially, $x = y = 0$

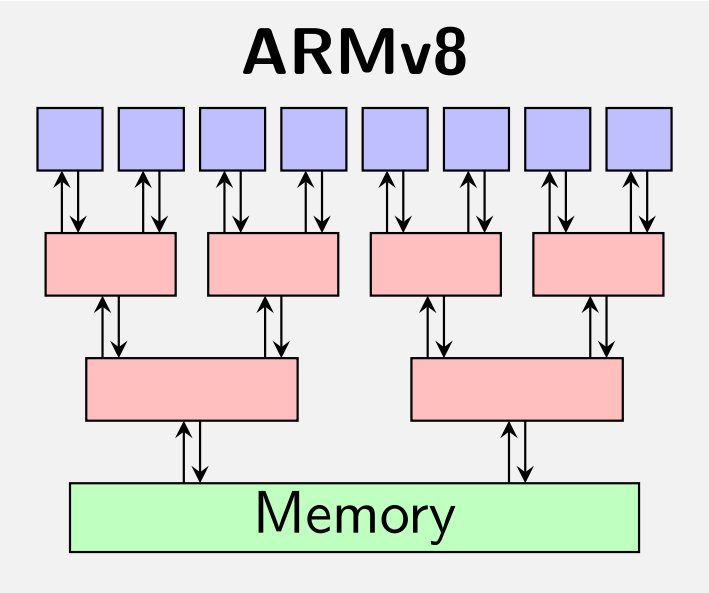
$$\begin{array}{ll|ll} x := 1; & & y := 1; & \\ a := y & \mathit{// 0} & b := x & \mathit{// 0} \end{array}$$



Load buffering (LB)

Initially, $x = y = 0$

$$\begin{array}{ll|ll} a := y; & \mathit{// 1} & b := x; & \mathit{// 1} \\ x := 1 & & y := 1 & \end{array}$$



Weak consistency in "real life"

- Messages may be delayed.

$$\begin{array}{l|l} MsgX := 1; & MsgY := 1; \\ a := MsgY; \quad // 0 & b := MsgX; \quad // 0 \end{array}$$

- Messages may be sent/received out of order.

$$\begin{array}{l|l} Email := 1; & a := Sms; \quad // 1 \\ Sms := 1; & b := Email; \quad // 0 \end{array}$$

Operational memory models

A simple concurrent programming language

Basic domains:

$r \in \mathsf{Reg}$	– Registers (local variables)
$x \in \mathsf{Loc}$	– Locations
$v \in \mathsf{Val}$	– Values including 0
$i \in \mathsf{Tid} = \{1, \ldots, N\}$	– Thread identifiers

Expressions and commands:

$$
\begin{aligned}
e &::= r \mid v \mid e + e \mid \ldots \\
c &::= \textbf{skip} \mid \textbf{if } e \textbf{ then } c \textbf{ else } c \mid \textbf{while } e \textbf{ do } c \mid \\
&\quad\; c \,;\, c \mid r := e \mid r := x \mid x := e \mid \\
&\quad\; r := \textbf{FAA}(x, e) \mid r := \textbf{CAS}(x, e, e) \mid \textbf{fence}
\end{aligned}
$$

Programs, $P : \mathsf{Tid} \to \mathsf{Cmd}$, written as $P = c_1 \| \ldots \| c_N$

Basic set up

Thread subsystem

- Thread-local steps: $c, s \xrightarrow{l} c', s'$.
- Interpret sequential programs.
- Lift them to program steps: $P, S \xrightarrow{i:l} P', S'$.

Storage subsystem (defined by the memory model)

- Describe the effect of memory accesses and fences.
- $M \xrightarrow{i:l} M'$ where M is the state of the storage subsystem.

Linking the two

- Either the thread or the storage subsystem make an internal step, ε; or they make matching $i{:}l$ steps.
- $P, S, M \Rightarrow P', S', M'$.

The thread subsystem

Store: $s : \mathrm{Reg} \to \mathrm{Val}$ (Initial store: $s_0 \triangleq \lambda r.\ 0$)

State: $\langle c, s \rangle \in \mathrm{Command} \times \mathrm{Store}$

Transitions:

$$\frac{}{\textbf{skip}; c, s \xrightarrow{\varepsilon} c, s} \qquad \frac{c_1, s \xrightarrow{l} c_1', s'}{c_1; c_2, s \xrightarrow{l} c_1'; c_2, s'} \qquad \frac{s' = s[r \mapsto s(e)]}{r := e, s \xrightarrow{\varepsilon} \textbf{skip}, s'}$$

$$\frac{l = \mathtt{R}(x, v)}{r := x, s \xrightarrow{l} \textbf{skip}, s[r \mapsto v]} \qquad \frac{l = \mathtt{W}(x, s(e))}{x := e, s \xrightarrow{l} \textbf{skip}, s}$$

$$\frac{s(e) \neq 0}{\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, s \xrightarrow{\varepsilon} c_1, s} \qquad \frac{s(e) = 0}{\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, s \xrightarrow{\varepsilon} c_2, s}$$

$$\frac{}{\textbf{while}\ e\ \textbf{do}\ c, s \xrightarrow{\varepsilon} \textbf{if}\ e\ \textbf{then}\ (c; \textbf{while}\ e\ \textbf{do}\ c)\ \textbf{else skip}, s}$$

The thread subsystem: RMW and fence commands

Fetch-and-add:

$$\frac{l = \mathtt{U}(x, v, v + s(e))}{r := \textbf{FAA}(x, e), s \xrightarrow{l} \textbf{skip}, s[r \mapsto v]}$$

Compare-and-swap:

$$\frac{l = \mathtt{R}(x, v) \qquad v \neq s(e_r)}{r := \textbf{CAS}(x, e_r, e_w), s \xrightarrow{l} \textbf{skip}, s[r \mapsto 0]}$$

$$\frac{l = \mathtt{U}(x, s(e_r), s(e_w))}{r := \textbf{CAS}(x, e_r, e_w), s \xrightarrow{l} \textbf{skip}, s[r \mapsto 1]}$$

Fence:

$$\frac{}{\textbf{fence}, s \xrightarrow{\mathtt{F}} \textbf{skip}, s}$$

Lifting to concurrent programs

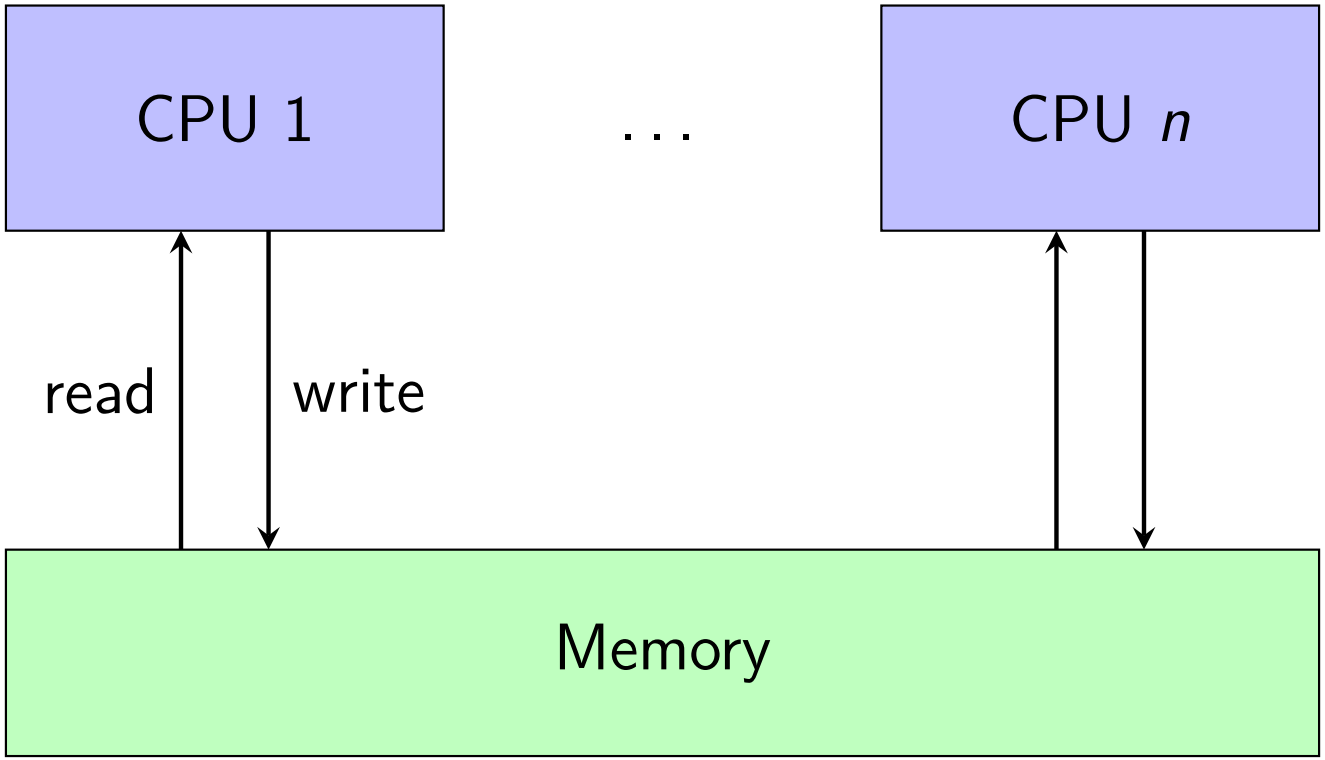
State: $\langle P, S \rangle \in \text{Program} \times (\text{Tid} \to \text{Store})$

- Initial stores: $S_0 \triangleq \lambda i.\ s_0$
- Initial state: $\langle P, S_0 \rangle$

Transition:

$$\frac{P(i), S(i) \xrightarrow{l} c, s}{P, S \xrightarrow{i:l} P[i \mapsto c], S[i \mapsto s]}$$

SC storage subsystem



SC storage subsystem

Machine state: $M : \mathsf{Loc} \to \mathsf{Val}$

- Maps each location to its value.
- Initial state: $M_0 \triangleq \lambda x.\ 0$
  (*i.e.*, the memory that maps every location to 0)

Transitions:

$$\frac{l = \mathtt{W}(x, v)}{M \xrightarrow{i:l} M[x \mapsto v]} \qquad \frac{l = \mathtt{R}(x, v) \qquad M(x) = v}{M \xrightarrow{i:l} M}$$

$$\frac{l = \mathtt{U}(x, v_r, v_w) \qquad M(x) = v_r}{M \xrightarrow{i:l} M[x \mapsto v_w]} \qquad \frac{l = \mathtt{F}}{M \xrightarrow{i:l} M}$$

SC: Linking the thread and storage subsystems

SILENT

$$\frac{P, S \xrightarrow{i:\varepsilon} P', S'}{P, S, M \Rightarrow P', S', M}$$

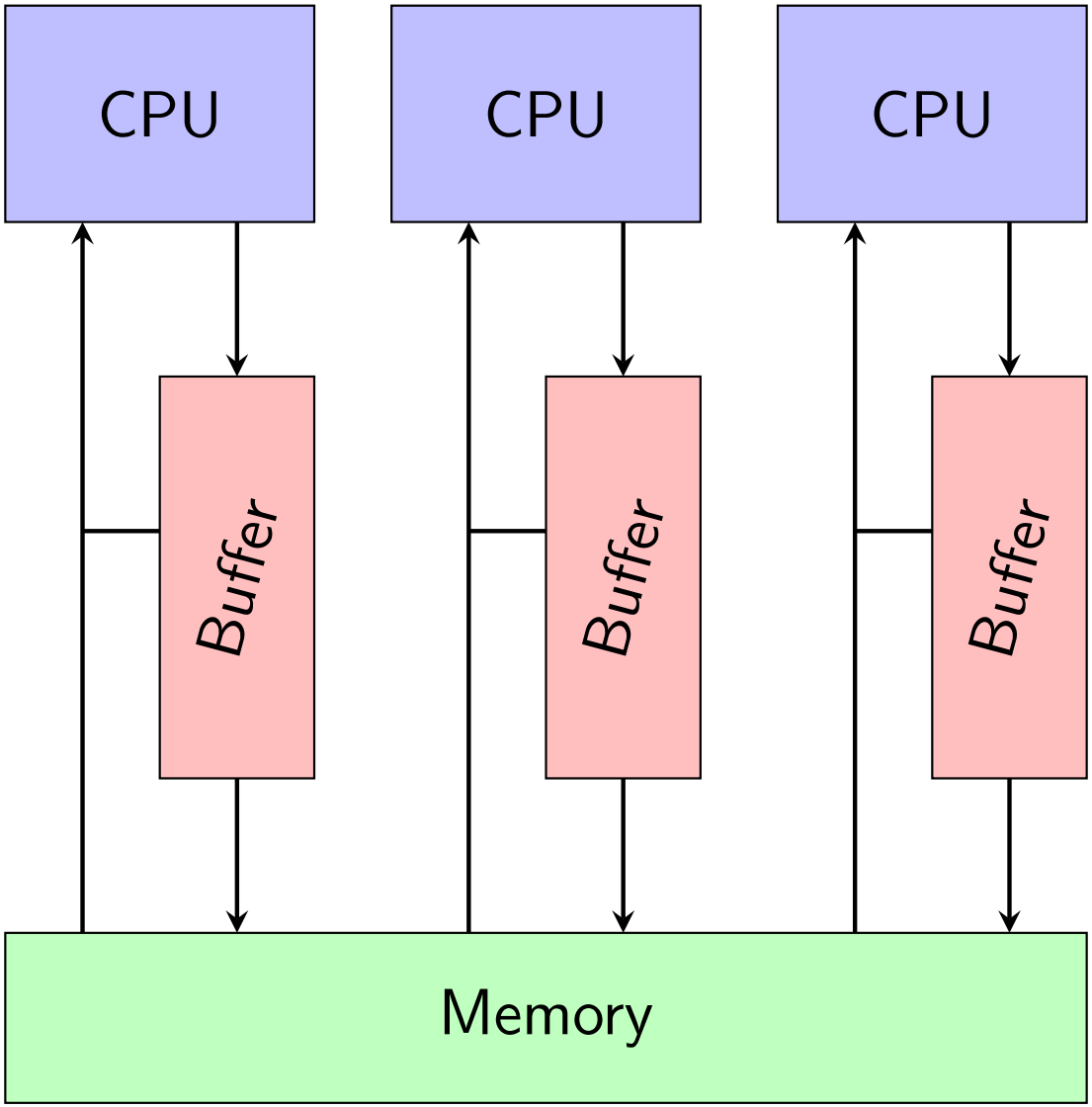
NON-SILENT

$$\frac{P, S \xrightarrow{i:l} P', S' \qquad M \xrightarrow{i:l} M'}{P, S, M \Rightarrow P', S', M'}$$

Definition (Allowed outcome)

- An *outcome* is a function $O : \mathrm{Tid} \to \mathrm{Store}$.
- An outcome O is *allowed* for a program P under SC if there exists M such that $P, S_0, M_0 \Rightarrow^* \mathbf{skip} \| \ldots \| \mathbf{skip}, O, M$.

TSO storage subsystem



TSO storage subsystem

The state consists of:

- A memory $M : \mathsf{Loc} \to \mathsf{Val}$
- A function $B : \mathsf{Tid} \to (\mathsf{Loc} \times \mathsf{Val})^*$
  assigning a *store buffer* to every thread.

Initial state: $\langle M_0, B_0 \rangle$ **where**

- $M_0 = \lambda x.\, 0$ (the memory maps 0 to every location)
- $B_0 = \lambda i.\, \epsilon$ (all store buffers are empty)

TSO storage subsystem transitions

WRITE

$$\frac{l = \mathtt{W}(x, v)}{M, B \xrightarrow{i:l} M, B[i \mapsto \langle x, v\rangle \cdot B(i)]}$$

PROPAGATE

$$\frac{B(i) = b \cdot \langle x, v\rangle}{M, B \xrightarrow{i:\varepsilon} M[x \mapsto v], B[i \mapsto b]}$$

READ

$$\frac{\begin{array}{c} l = \mathtt{R}(x, v) \\ B(i) = \langle x_n, v_n\rangle \cdot \ldots \cdot \langle x_2, v_2\rangle \cdot \langle x_1, v_1\rangle \\ M[x_1 \mapsto v_1][x_2 \mapsto v_2] \ldots [x_n \mapsto v_n](x) = v \end{array}}{M, B \xrightarrow{i:l} M, B}$$

RMW

$$\frac{l = \mathtt{U}(x, v_r, v_w) \qquad B(i) = \epsilon \qquad M(x) = v_r}{M, B \xrightarrow{i:l} M[x \mapsto v_w], B}$$

FENCE

$$\frac{l = \mathtt{F} \qquad B(i) = \epsilon}{M, B \xrightarrow{i:l} M, B}$$

TSO: linking thread and storage subsystems

SILENT-THREAD

$$\frac{P, S \xrightarrow{i:\varepsilon} P', S'}{P, S, M, B \Rightarrow P', S', M, B}$$

SILENT-STORAGE

$$\frac{M, B \xrightarrow{i:\varepsilon} M', B'}{P, S, M, B \Rightarrow P, S, M', B'}$$

NON-SILENT

$$\frac{P, S \xrightarrow{i:l} P', S' \qquad M, B \xrightarrow{i:l} M', B'}{P, S, M, B \Rightarrow P', S', M', B'}$$

Definition (Allowed outcome)

An outcome O is *allowed* for a program P under TSO if there exists M such that $P, S_0, M_0, B_0 \Rightarrow^* \mathbf{skip} \| \ldots \| \mathbf{skip}, O, M, B_0$.

Axiomatic memory models

An alternative way of defining the semantics

Declarative/axiomatic concurrency semantics

- Define the notion of a program *execution*
  (generalization of an execution trace)
- Map a program to a set of executions
- Define a *consistency* predicate on executions
- Semantics = set of consistent executions of a program

Exception: "catch-fire" semantics

- Existence of at least one "bad" consistent execution implies undefined behavior.

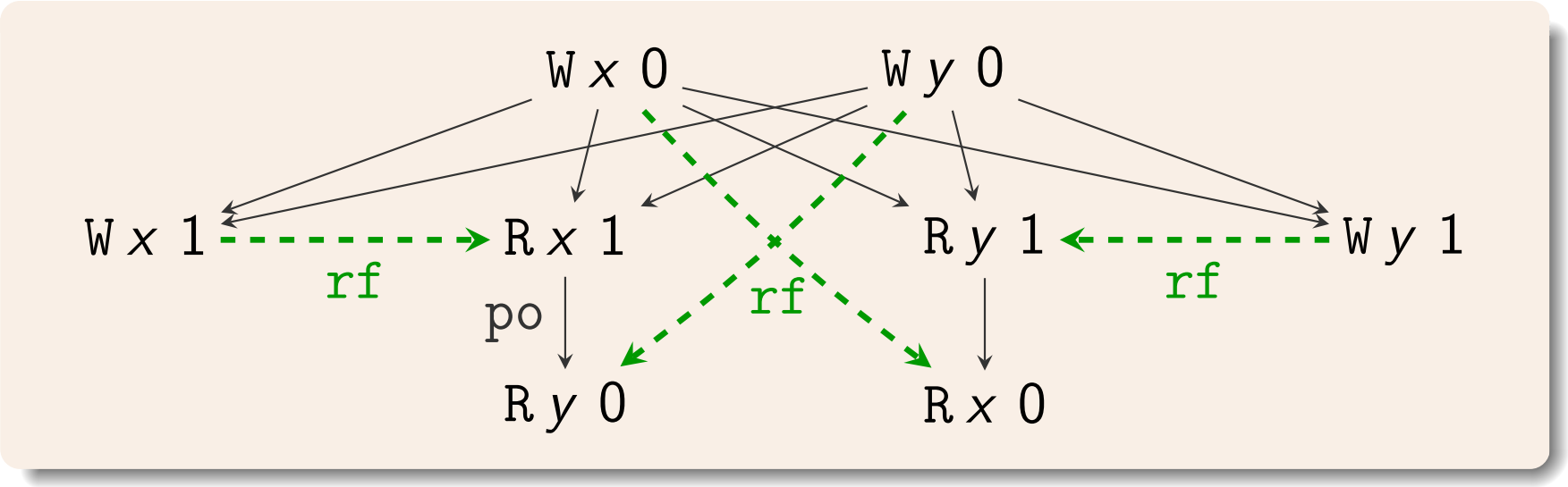
Executions

Events

- Reads, Writes, Updates, Fences

Relations

- Program order, po (also called "sequenced-before", sb)
- Reads-from, rf



Executions

Definition (Label)

A *label* has one of of the following forms:

$$\mathtt{R}\ x\ v_r \qquad \mathtt{W}\ x\ v_w \qquad \mathtt{U}\ x\ v_r\ v_w \qquad \mathtt{F}$$

where $x \in \mathsf{Loc}$ and $v_r, v_w \in \mathsf{Val}$.

Definition (Event)

An *event* is a triple $\langle id, i, l \rangle$ where

- $id \in \mathbb{N}$ is an event identifier,
- $i \in \mathsf{Tid} \cup \{0\}$ is a thread identifier, and
- l is a label.

Executions

Definition (Execution graph)

An *execution graph* is a tuple $\langle E, po, rf \rangle$ where:

- E is a finite set of events
- po ("*program order*") is a partial order on E
- rf ("*reads-from*") is a binary relation on E such that:
  - For every $\langle w, r \rangle \in rf$
    - $\mathtt{typ}(w) \in \{\mathtt{W}, \mathtt{U}\}$
    - $\mathtt{typ}(r) \in \{\mathtt{R}, \mathtt{U}\}$
    - $\mathtt{loc}(w) = \mathtt{loc}(r)$
    - $\mathtt{val_w}(w) = \mathtt{val_r}(r)$
  - rf^{-1} is a function

    (that is: if $\langle w_1, r \rangle, \langle w_2, r \rangle \in rf$ then $w_1 = w_2$)

Some notations

Let $G = \langle E, po, rf \rangle$ be an execution graph.

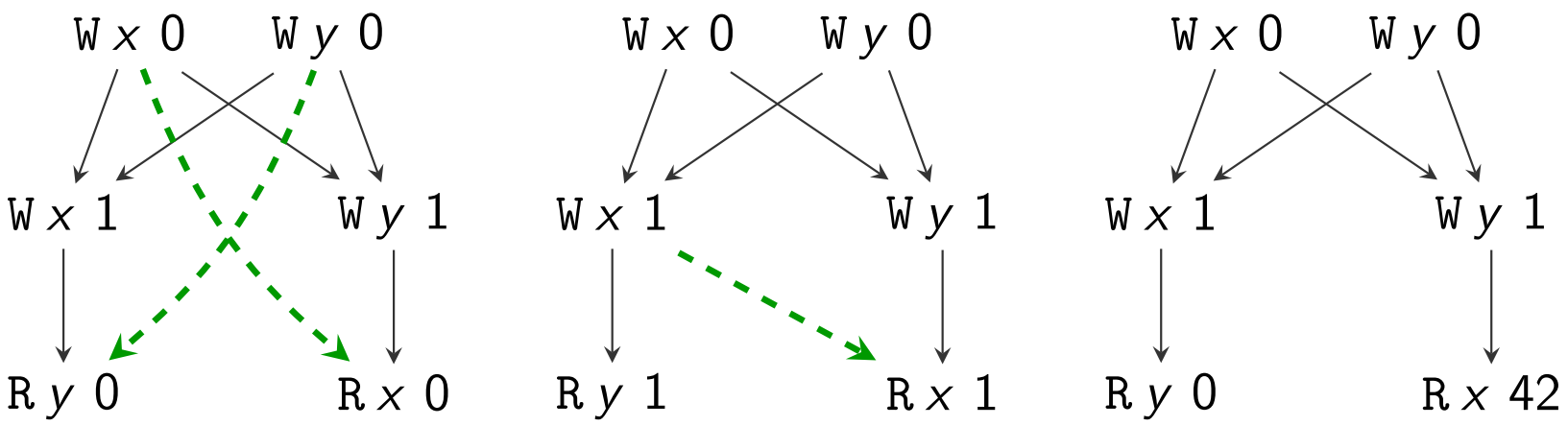
- $G.\mathtt{E} \triangleq E$
- $G.\mathrm{po} \triangleq po$
- $G.\mathtt{rf} \triangleq rf$
- $G.\mathtt{R} \triangleq \{r \in E \mid \mathtt{typ}(r) = \mathtt{R} \vee \mathtt{typ}(r) = \mathtt{U}\}$
- $G.\mathtt{W} \triangleq \{w \in E \mid \mathtt{typ}(w) = \mathtt{W} \vee \mathtt{typ}(w) = \mathtt{U}\}$
- $G.\mathtt{U} \triangleq \{u \in E \mid \mathtt{typ}(u) = \mathtt{U}\}$
- $G.\mathtt{F} \triangleq \{f \in E \mid \mathtt{typ}(f) = \mathtt{F}\}$
- $G.\mathtt{R}_x \triangleq G.\mathtt{R} \cap \{r \in E \mid \mathtt{loc}(r) = x\}$
- ...

Mapping programs to executions: Example

Store buffering (SB)

$$x = y = 0$$

$$\begin{array}{c||c} x := 1 & y := 1 \\ a := y & b := x \end{array}$$



Consistency predicate

Let X be some consistency predicate (on execution graphs)

Definition (Allowed outcome under a declarative model)

An outcome O is *allowed* for a program P under X if there exists an execution graph G such that:

- G is an execution graph of P with outcome O.
- G is X-consistent.

Exception: "catch-fire" semantics

... or if there exists an execution graph G such that:

- G is an execution graph of P.
- G is X-consistent.
- G is "bad".

Completeness

The most basic consistency condition:

Definition (Completeness)

An execution graph G is called *complete* if

$$codom(G.\mathtt{rf}) = G.\mathtt{R}$$

i.e., *every* read reads from *some* write.

Sequential consistency

the result of any execution is the same as if the operations of all the processors were executed in some sequential order, respecting the order specified by the program *[Lamport, 1979]*

Sequential consistency [Lamport]

Definition

Let $\mathtt{sc}$ be a total order on $G.\mathtt{E}$. G is called *SC-consistent* wrt $\mathtt{sc}$ if the following hold:

- If $\langle a, b\rangle \in G.\mathtt{po}$ then $\langle a, b\rangle \in \mathtt{sc}$.
- If $\langle a, b\rangle \in G.\mathtt{rf}$ then $\langle a, b\rangle \in \mathtt{sc}$ and there does not exist $c \in G.\mathtt{W}_{\mathtt{loc}(b)}$ such that $\langle a, c\rangle \in \mathtt{sc}$ and $\langle c, b\rangle \in \mathtt{sc}$.

Definition

An execution graph G is called *SC-consistent* if the following hold:

- G is complete.
- G is SC-consistent wrt some total order $\mathtt{sc}$ on $G.\mathtt{E}$.

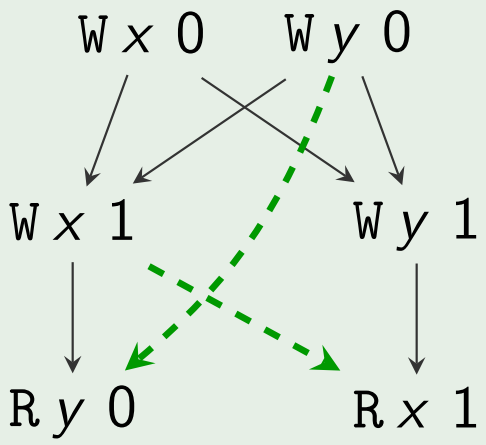
SB example

Store buffering (SB)

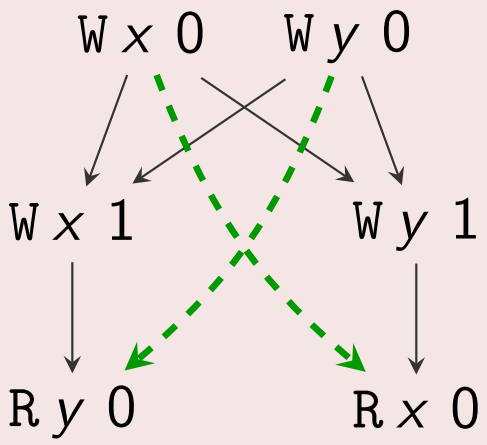
$$x = y = 0$$

$$\begin{array}{c||c} x := 1 & y := 1 \\ a := y & b := x \end{array}$$

Allowed



Forbidden



Sequential consistency (Alternative)

Definition (Modification order (aka coherence order))

$\mathtt{mo}$ is called a *modification order* for an execution graph G if $\mathtt{mo} = \bigcup_{x \in \mathsf{Loc}} \mathtt{mo}_x$ where each $\mathtt{mo}_x$ is a total order on $G.\mathtt{W}_x$.

Definition (Alternative SC definition)

An execution graph G is called *SC-consistent* if the following hold:

- G is complete
- There exists a modification order $\mathtt{mo}$ for G such that $G.\mathrm{po} \cup G.\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic where:
  - $\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathrm{id}$ (from-reads / reads-before)

Equivalence

Theorem

The two SC definitions are equivalent.

Proof (sketch).

Lamport SC $\Rightarrow$ alternative SC:

- Take $\mathtt{mo}_x \triangleq [\mathtt{W}_x]; \mathtt{sc}; [\mathtt{W}_x]$.
- Then, $G.\mathrm{po} \cup G.\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb} \subseteq \mathtt{sc}$.

Alternative SC $\Rightarrow$ Lamport SC:

- Take $\mathtt{sc}$ to be any total order extending $G.\mathrm{po} \cup G.\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$. $\square$

Relaxing sequential consistency

- SC is very expensive to implement in hardware.
- It also forbids various optimizations that are sound for sequential code.

What most hardware guarantee and compilers preserve is "SC-per-location" (aka *coherence*).

Definition

An execution graph G is called *coherent* if the following hold:

- G is complete
- For every location x, there exists a total order $\mathtt{sc}_x$ on all accesses to x such that:
  - If $\langle a, b\rangle \in [\mathtt{RW}_x]; G.\mathtt{po}; [\mathtt{RW}_x]$ then $\langle a, b\rangle \in \mathtt{sc}_x$
  - If $\langle a, b\rangle \in [\mathtt{W}_x]; G.\mathtt{rf}; [\mathtt{R}_x]$ then $\langle a, b\rangle \in \mathtt{sc}_x$ and there does not exist $c \in G.\mathtt{W}_x$ such that $\langle a, c\rangle \in \mathtt{sc}_x$ and $\langle c, b\rangle \in \mathtt{sc}_x$.

Alternative definition of coherence I

SC: $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
COH: $\mathtt{po}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic

Definition

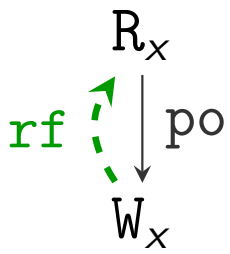
Let $\mathtt{mo}$ be a modification order for an execution graph G. G is called *coherent wrt* $\mathtt{mo}$ if $G.\mathtt{po}|_{\mathtt{loc}} \cup G.\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic (where $\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathrm{id}$).

Theorem

An execution graph G is coherent iff the following hold:

- *G is complete*
- *G is coherent wrt some modification order $\mathtt{mo}$ for G.*

“Bad patterns” I



$a := x$ // 1
$x := 1$

no-future-read

$r := \mathbf{CAS}(x, 1, 1)$ // 1

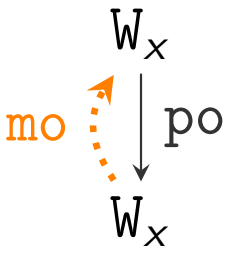
rmw-1

Recall:

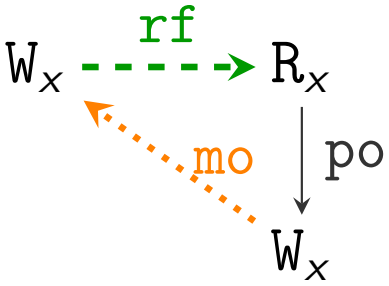
- $\mathtt{W}$ is either a write or an RMW.
- $\mathtt{R}$ is either a read or an RMW.

“Bad patterns” II

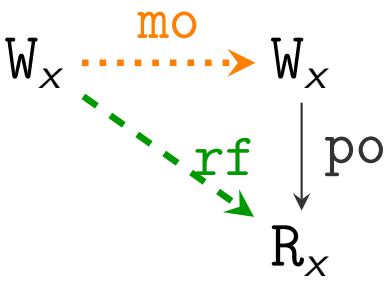
$$\begin{array}{l||l} x := 1 & a := x \mathbin{/\!/} 2 \\ x := 2 & a := x \mathbin{/\!/} 1 \end{array}$$



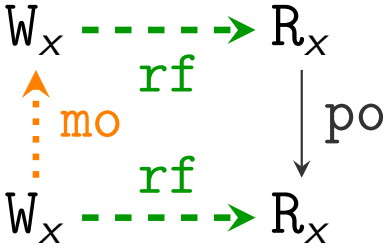
coherence-ww



coherence-rw

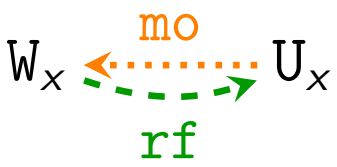


coherence-wr

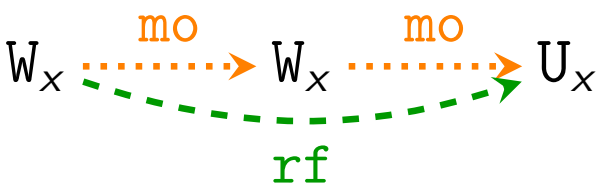


coherence-rr

“Bad patterns” III



rmw-2



atomicity

In coherent executions, an RMW event may only read from its immediate mo-predecessor.

Alternative definition of coherence II

Theorem

Let $\mathtt{mo}$ *be a modification order for an execution graph* G*.*
G *is coherent wrt* $\mathtt{mo}$ *iff the following hold:*

- $\mathtt{rf};\mathtt{po}$ *is irreflexive.* *(no-future-read)*
- $\mathtt{mo};\mathtt{po}$ *is irreflexive.* *(coherence-ww)*
- $\mathtt{mo};\mathtt{rf};\mathtt{po}$ *is irreflexive.* *(coherence-rw)*
- $\mathtt{rf}^{-1};\mathtt{mo};\mathtt{po}$ *is irreflexive.* *(coherence-wr)*
- $\mathtt{rf}^{-1};\mathtt{mo};\mathtt{rf};\mathtt{po}$ *is irreflexive.* *(coherence-rr)*
- $\mathtt{rf}$ *is irreflexive.* *(rmw-1)*
- $\mathtt{mo};\mathtt{rf}$ *is irreflexive.* *(rmw-2)*
- $\mathtt{rf}^{-1};\mathtt{mo};\mathtt{mo}$ *is irreflexive.* *(rmw-atomicity)*

Examples (aka "litmus tests")

Coherence test

$$x = 0$$

$$\begin{array}{l || l} x := 1 & x := 2 \\ a := x \;\; /\!/ \; 2 & b := x \;\; /\!/ \; 1 \end{array}$$

Store buffering

$$x = y = 0$$

$$\begin{array}{l || l} x := 1 & y := 1 \\ a := y \;\; /\!/ \; 0 & b := x \;\; /\!/ \; 0 \end{array}$$

Atomicity

Parallel increment

$$x = 0$$
$$a := \textbf{FAA}(x, 1) \;\|\; b := \textbf{FAA}(x, 1)$$

Guarantees that $a = 1 \vee b = 1$.

Can we implement locks in this semantics?

Spinlock implementation

lock(l) :	**unlock**(l) :
$r := 0;$	$l := 0$
while $\neg r$ **do**	
$\quad r := \textbf{CAS}(l, 0, 1)$	

Implementing locks?

Under COH, the spinlock implementation does not guarantee mutual exclusion.

Spinlock implementation

lock(l) :	**unlock**(l) :
$r := 0$;	$l := 0$
while $\neg r$ **do**	
$\quad r :=$ **CAS**$(l, 0, 1)$	

Lock example

lock(l)	**lock**(l)
$x := 1$	$y := 1$
$a := y$ *// 0*	$b := x$ *// 0*
unlock(l)	**unlock**(l)

Message passing

More generally, COH is often too weak:

$$x = y = 0$$

$$\begin{array}{l} x := 42; \\ y := 1 \end{array} \quad \Bigg\| \quad \begin{array}{l} a := y; \\ \textbf{while } \neg a \textbf{ do } a := y; \\ b := x \quad \textit{// 0} \end{array}$$

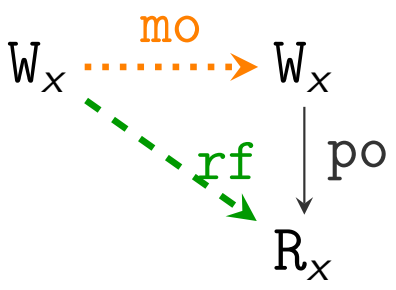
$$x = y = 0$$

$$\begin{array}{l} x := 42; \\ y := 1 \end{array} \quad \Bigg\| \quad \begin{array}{l} a := y; \quad \textit{// 1} \\ b := x \quad \textit{// 0} \end{array}$$

MP is a common programming idiom.

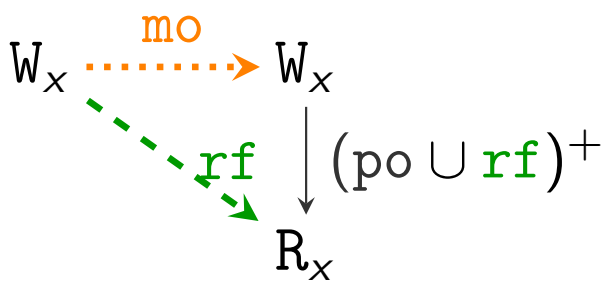
How can we disallow the weak behavior?

Supporting message passing



coherence-wr

$\rightsquigarrow$



coherence-wr

Solution:

- Strengthen the notion of an "observed" write.
- In other words, make `rf`-edges "synchronizing."

Release/acquire (RA) memory model

SC: $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
COH: $\mathtt{po}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
RA: $(\mathtt{po} \cup \mathtt{rf})^{+}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic

Definition

Let $\mathtt{mo}$ be a modification order for an execution graph G.
G is called RA-*consistent wrt* $\mathtt{mo}$ if $(\mathtt{po} \cup \mathtt{rf})^{+}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic for some modification order $\mathtt{mo}$ for G (where $\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathrm{id}$).

Definition

An execution graph G is RA-*consistent* if the following hold:

- G is complete
- G is RA-consistent wrt some modification order $\mathtt{mo}$ for G.

Alternative definition of RA consistency

Theorem

Let $\mathtt{mo}$ *be a modification order for an execution graph* G*.* G *is* RA-*consistent wrt* $\mathtt{mo}$ *iff the following hold:*

- $(\mathrm{po} \cup \mathtt{rf})^+$ *is irreflexive.* *(no-future-read)*
- $\mathtt{mo}; (\mathrm{po} \cup \mathtt{rf})^+$ *is irreflexive.* *(coherence-ww)*
- $\mathtt{rf}^{-1}; \mathtt{mo}; (\mathrm{po} \cup \mathtt{rf})^+$ *is irreflexive.* *(coherence-wr)*
- $\mathtt{rf}^{-1}; \mathtt{mo}; \mathtt{mo}$ *is irreflexive.* *(rmw-atomicity)*

The C/C++11 memory model

Mixing the models

$$\mathsf{COH} < \mathsf{RA} < \mathsf{SC}$$

- Revisit the MP idiom:

$$\begin{array}{l} x := 42 \\ y := 1 \end{array} \quad \Bigg\| \quad \begin{array}{l} a := y \\ \textbf{while } \neg a \textbf{ do } a := y \\ b := x \quad \mathit{// 0} \end{array}$$

- We only need the last read of y to synchronize.
- Idea: introduce *access modes*.

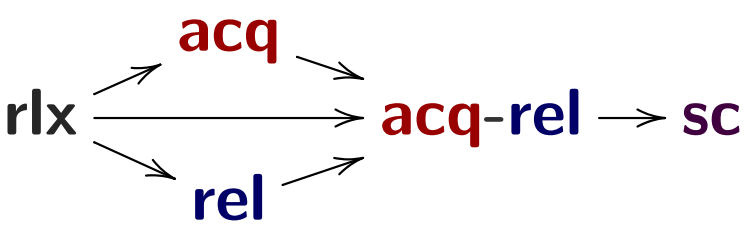
$$\begin{array}{l} x :=_{\mathbf{rlx}} 42 \\ y :=_{\mathbf{rel}} 1 \end{array} \quad \Bigg\| \quad \begin{array}{l} a := y_{\mathbf{rlx}} \\ \textbf{while } \neg a \textbf{ do } a := y \\ a := y_{\mathbf{acq}} \\ b := x_{\mathbf{rlx}} \quad \mathit{// 0} \end{array}$$

Happens-before

Each memory accesses has a *mode*:

- Reads: **rlx**, **acq**, or **sc**
- Writes: **rlx**, **rel**, or **sc**
- RMWs: **rlx**, **acq**, **rel**, **acq-rel**, or **sc**

"Strength" order $\sqsubset$ is given by:



Synchronization:

$$G.\mathtt{sw} = [\mathtt{W}^{\sqsupseteq \mathbf{rel}}]; G.\mathtt{rf}; [\mathtt{R}^{\sqsupseteq \mathbf{acq}}]$$

Happens-before:

$$G.\mathtt{hb} = (G.\mathrm{po} \cup G.\mathtt{sw})^{+}$$

Towards C/C++11 memory model

SC: $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
COH: $\mathtt{po}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
RA: $(\mathtt{po} \cup \mathtt{rf})^{+}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
C11: $\mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic

Definition

Let $\mathtt{mo}$ be a modification order for an execution graph G. G is called C11-*consistent wrt* $\mathtt{mo}$ if $\mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic (where $\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathrm{id}$).

Definition

An execution graph G is C11-consistent if the following hold:

- G is complete
- G is C11-consistent wrt some modification order $\mathtt{mo}$ for G.

The C/C++11 memory model

non-atomic	$\sqsubset$	relaxed	$\sqsubset$	release/ acquire	$\sqsubset$	sc

The full C/C++11 is more general:

- Non-atomics for non-racy code (the default!)
- Four types of fences for fine grained control
- SC accesses to ensure sequential consistency if needed
- More elaborate definition of `sw` ("release sequences")

C11 model through examples

C11 model through examples

1

```
        int a = 0;
        int x = 0;
a = 42; ‖ if(x == 1){
x = 1;  ‖     print(a);
        ‖ }
```

C11 model through examples

1

```
        int a = 0;
        int x = 0;
a = 42; | if(x == 1){
x = 1;  |   print(a);
        | }
```

race

C11 model through examples

1

```
        int a = 0;
        int x = 0;
a = 42; | if(x == 1){
x = 1;  |   print(a);
        | }
```

race

2

```
        int a = 0;
    atomic_int x = 0;
a = 42;     | if(x_rlx == 1){
x_rlx = 1;  |     print(a);
            | }
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42;  ‖ if(x == 1){
x = 1;   ‖ race   print(a);
         ‖ }
```

2

```
int a = 0;
atomic_int x = 0;
a = 42;      ‖ if(x_rlx == 1){
x_rlx = 1;   ‖ race   print(a);
             ‖ }
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42;  ‖ if(x == 1){
x = 1;   ‖ race   print(a);
         ‖ }
```

2

```
int a = 0;
atomic_int x = 0;
a = 42;     ‖ if(x_rlx == 1){
x_rlx = 1;  ‖ race   print(a);
            ‖ }
```

3

```
int a = 0;
atomic_int x = 0;
a = 42;     ‖ if(x_acq == 1){
x_rel = 1;  ‖     print(a);
            ‖ }
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42;  ‖ if(x == 1){
x = 1;   ‖ race   print(a);
         ‖ }
```

2

```
int a = 0;
atomic_int x = 0;
a = 42;      ‖ if(x_rlx == 1){
x_rlx = 1;   ‖ race   print(a);
             ‖ }
```

3

```
int a = 0;
atomic_int x = 0;
a = 42;      ‖ if(x_acq == 1){
x_rel = 1;   ‖ rf   print(a);
             ‖ }
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42;  ‖ if(x == 1){
x = 1;   ‖ race   print(a);
         ‖ }
```

2

```
int a = 0;
atomic_int x = 0;
a = 42;      ‖ if(x_rlx == 1){
x_rlx = 1;   ‖ race   print(a);
             ‖ }
```

3

```
int a = 0;
atomic_int x = 0;
a = 42;      ‖ if(x_acq == 1){
x_rel = 1;   ‖ rf sw   print(a);
             ‖ }
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42;  ‖ if(x == 1){
x = 1;   ‖ race   print(a);
         ‖ }
```

2

```
int a = 0;
atomic_int x = 0;
a = 42;   ‖ if(x_rlx == 1){
x_rlx = 1; ‖ race  print(a);
           ‖ }
```

3

```
int a = 0;
atomic_int x = 0;
a = 42;   ‖ if(x_acq == 1){
x_rel = 1; ‖ rf sw  print(a);
           ‖ }
```

4

```
int a = 0;
atomic_int x = 0;
a = 42;        ‖ if(x_rlx == 1){
fence_rel;     ‖     fence_acq;
x_rlx = 1;     ‖     print(a);
               ‖ }
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42;  ‖ if(x == 1){
x = 1;   ‖ race   print(a);
         ‖ }
```

2

```
int a = 0;
atomic_int x = 0;
a = 42;   ‖ if(x_rlx == 1){
x_rlx = 1; ‖ race  print(a);
           ‖ }
```

3

```
int a = 0;
atomic_int x = 0;
a = 42;   ‖ if(x_acq == 1){
x_rel = 1; ‖ rf sw  print(a);
           ‖ }
```

4

```
int a = 0;
atomic_int x = 0;
a = 42;        ‖ if(x_rlx == 1){
fence_rel;     ‖ rf   fence_acq;
x_rlx = 1;     ‖      print(a);
               ‖ }
```

C11 model through examples

1

```
int a = 0;
int x = 0;
a = 42;  ‖ if(x == 1){
x = 1;   ‖ race   print(a);
         ‖ }
```

2

```
int a = 0;
atomic_int x = 0;
a = 42;   ‖ if(x_rlx == 1){
x_rlx = 1; ‖ race  print(a);
           ‖ }
```

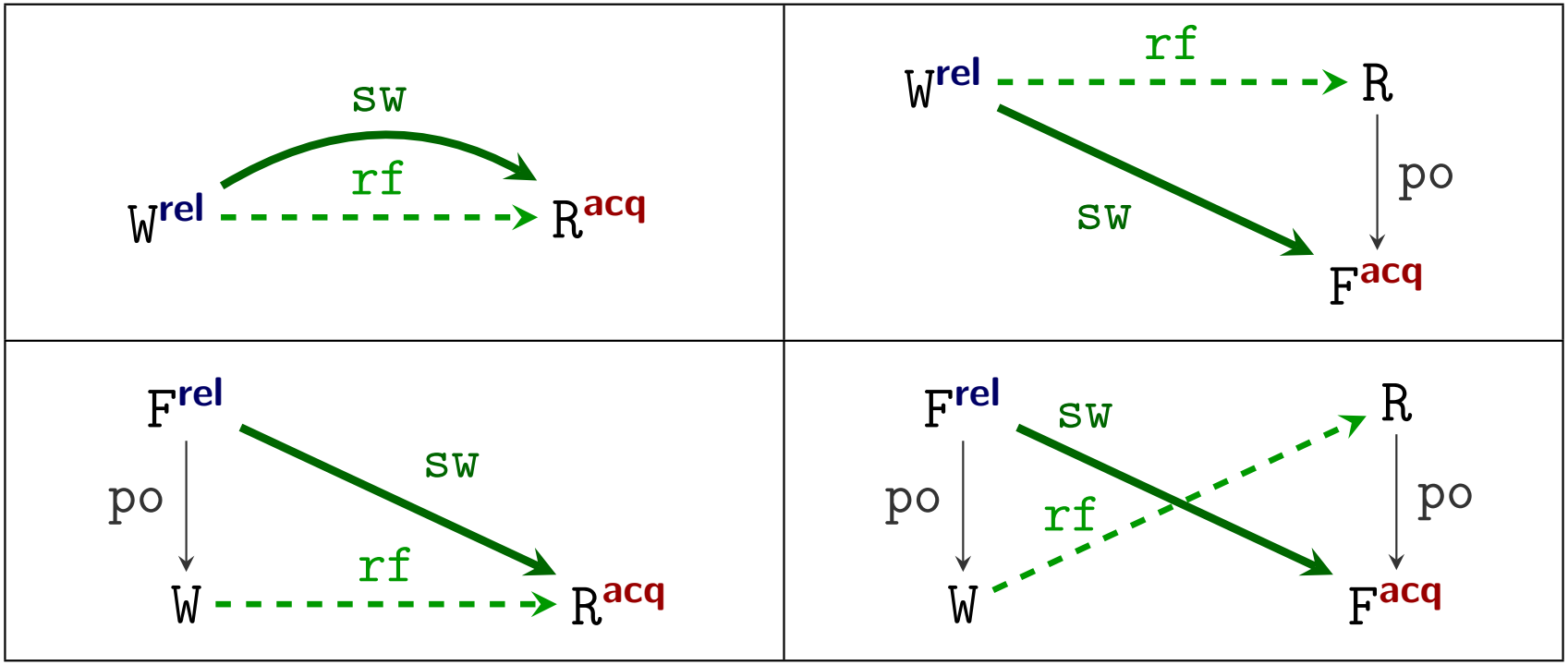
3

```
int a = 0;
atomic_int x = 0;
a = 42;   ‖ if(x_acq == 1){
x_rel = 1; ‖ rf sw  print(a);
           ‖ }
```

4

```
int a = 0;
atomic_int x = 0;
a = 42;        ‖ if(x_rlx == 1){
fence_rel;     ‖ rf sw  fence_acq;
x_rlx = 1;     ‖        print(a);
               ‖ }
```

The "synchronizes-with" relation

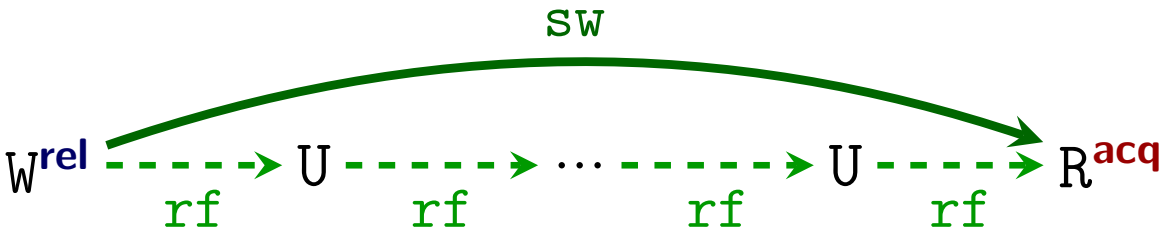


$$\mathtt{sw} \triangleq ([\mathtt{W}^{\sqsupseteq \mathbf{rel}}] \cup [\mathtt{F}^{\sqsupseteq \mathbf{rel}}]; \mathtt{po}); \mathtt{rf}; ([\mathtt{R}^{\sqsupseteq \mathbf{acq}}] \cup \mathtt{po}; [\mathtt{F}^{\sqsupseteq \mathbf{acq}}])$$

Fence modes

Release sequences (RMW's)

$x_{\mathbf{rlx}} := 42;$ $y_{\mathbf{rel}} := 1$	$a := \mathbf{FAI}_{\mathbf{rlx}}(y);$ // 1	$b := y_{\mathbf{acq}};$ // 2 $c := x_{\mathbf{rlx}};$ // 0



$$\mathtt{sw} \triangleq ([\mathtt{W}^{\sqsupseteq \mathbf{rel}}] \cup [\mathtt{F}^{\sqsupseteq \mathbf{rel}}]; \mathtt{po}); \mathtt{rf}^{+}; ([\mathtt{R}^{\sqsupseteq \mathbf{acq}}] \cup \mathtt{po}; [\mathtt{F}^{\sqsupseteq \mathbf{acq}}])$$

"Catch-fire" semantics

Definition (Race in C11)

Given a C11-execution graph G, we say that two events a, b C11-*race* in G if the following hold:

- $a \neq b$
- $\mathtt{loc}(a) = \mathtt{loc}(b)$
- $\{\mathtt{typ}(a), \mathtt{typ}(b)\} \cap \{\mathtt{W}, \mathtt{U}\} \neq \emptyset$
- $\mathbf{na} \in \{\mathtt{mod}(a), \mathtt{mod}(b)\}$
- $\langle a, b\rangle \notin \mathtt{hb}$ and $\langle b, a\rangle \notin \mathtt{hb}$

G is called C11-*racy* if some a, b C11-race in G.

C11 consistency

Definition

Let $\mathtt{mo}$ be a modification order for an execution graph G.
G is called C11-*consistent wrt* $\mathtt{mo}$ if:

- $\mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic (where $\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathrm{id}$).
- ...**sc**... ?

Definition

An execution graph G is C11-consistent if the following hold:

- G is complete
- G is C11-consistent wrt some modification order $\mathtt{mo}$ for G.

SC conditions

- The most involved part of the model, due to the possible mixing of different access modes to the same location.
- Changed in C++20
- *If there is no mixing of SC and non-SC accesses*, then additionally require acyclicity of $\mathtt{hb} \cup \mathtt{mo}_{\mathbf{sc}} \cup \mathtt{rb}_{\mathbf{sc}}$.

Further reading:

- Overhauling SC atomics in C11 and OpenCL. Mark Batty, Alastair F. Donaldson, John Wickerson, POPL 2016.
- Repairing sequential consistency in C/C++11. Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, Derek Dreyer, PLDI 2017.

Repaired SC condition for fences

$$\mathtt{eco} \triangleq (\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^+ \qquad \text{(extended coherence order)}$$

$$\mathtt{psc}_{\mathtt{F}} \triangleq [\mathtt{F}^{\mathbf{sc}}]; (\mathtt{hb} \cup \mathtt{hb}; \mathtt{eco}; \mathtt{hb}); [\mathtt{F}^{\mathbf{sc}}] \qquad \text{(SC fence order)}$$

Condition on SC fences

$\mathtt{psc}_{\mathtt{F}}$ is acyclic

Example: SB with fences

$$x = y = 0$$

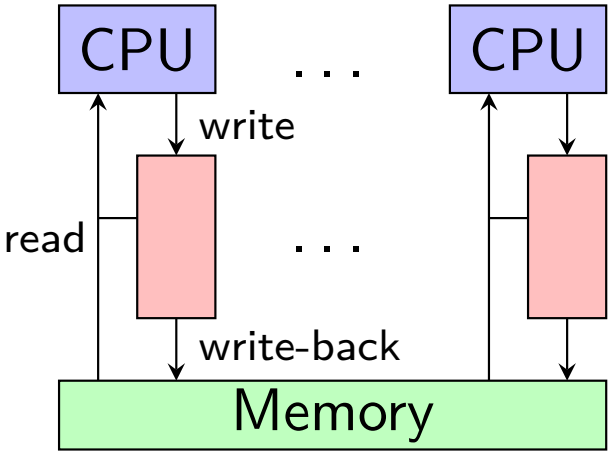
$x_{\mathbf{rlx}} := 1;$	$y_{\mathbf{rlx}} := 1;$
fence(**sc**);	**fence**(**sc**);
$a := y_{\mathbf{rlx}};$ // 0	$b := x_{\mathbf{rlx}};$ // 0

✗ behavior disallowed

Reduction from RA to SC

Reduction to SC (robustness)

For TSO, it suffices to have a fence between every racy write & subsequent racy read.



For RA, we need more fences. Recall the IRIW example:

Independent reads of independent writes (IRIW)

$$x = y = 0$$

$$x := 1 \;\Big\|\; \begin{array}{ll} a := x; & \mathit{//1} \\ b := y & \mathit{//0} \end{array} \;\Big\|\; \begin{array}{ll} c := y; & \mathit{//1} \\ d := x & \mathit{//0} \end{array} \;\Big\|\; y := 1$$

What is the semantics of SC fences?

From C11, we had:

$$\mathtt{eco} \triangleq (\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^+ \qquad \text{(extended coherence order)}$$

$$\mathtt{psc}_{\mathtt{F}} \triangleq [\mathtt{F}^{\mathtt{sc}}]; (\mathtt{hb} \cup \mathtt{hb}; \mathtt{eco}; \mathtt{hb}); [\mathtt{F}^{\mathtt{sc}}] \qquad \text{(partial SC fence order)}$$

and required that $\mathtt{psc}_{\mathtt{F}}$ is acyclic.

That is,

Definition (RA consistency with fences)

An execution graph G is RA-*consistent* iff there exists some modification order $\mathtt{mo}$ for G such that:

- G is complete,
- $(\mathrm{po} \cup \mathtt{rf})^+|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic, and
- $\mathtt{psc}_{\mathtt{F}}$ is acyclic.

Alternative definition of RA consistency

Theorem

An execution graph G is RA*-consistent iff there exists a total order* $\mathtt{sc}$ *on* $G.\mathtt{F}^{sc}$ *and a modification order* $\mathtt{mo}$ *for* G *such that:*

- *G is complete,*
- $(\mathrm{po} \cup \mathtt{rf} \cup \mathtt{sc})^{+}$ *is irreflexive, and*
- $(\mathrm{po} \cup \mathtt{rf} \cup \mathtt{sc})^{*}; \mathtt{eco}$ *is irreflexive.*

Simple reduction theorem

Theorem

Let G be an RA*-consistent execution graph. If*

- *For every G-racy events a, b, if $\langle a, b\rangle \in (G.\mathtt{po} \cup G.\mathtt{rf})^+$, then $\langle a, c\rangle, \langle c, b\rangle \in (G.\mathtt{po} \cup G.\mathtt{rf})^+$ for some fence event c.*

Then, G is SC-consistent.

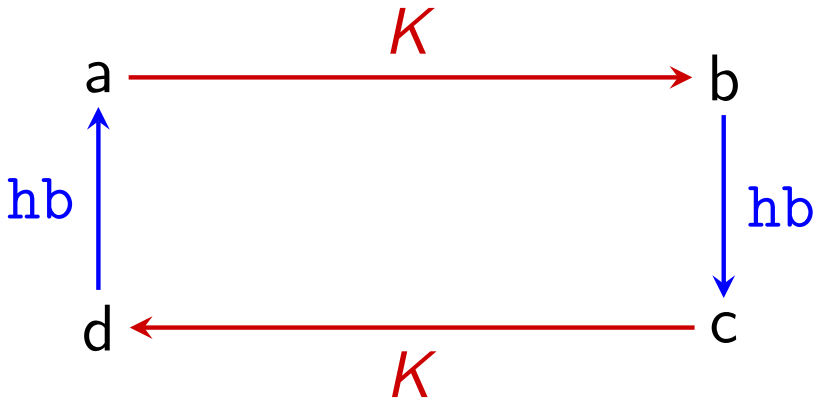
Proof of the simple reduction theorem (1/2)

Recall:

- Recall SC-consistency : $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic.
- Let $\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{rf} \cup \mathtt{sc})^+$ and $K \triangleq (\mathtt{mo} \cup \mathtt{rb}) \setminus \mathtt{hb}$.
- It suffices to prove : $\mathtt{hb} \cup K$ is acyclic.

Consider minimal cycle in $(\mathtt{hb} \cup K)$.

- Cycles with ≤ 1 K-edges disallowed by RA consistency.
- Cycle with two K-edges:



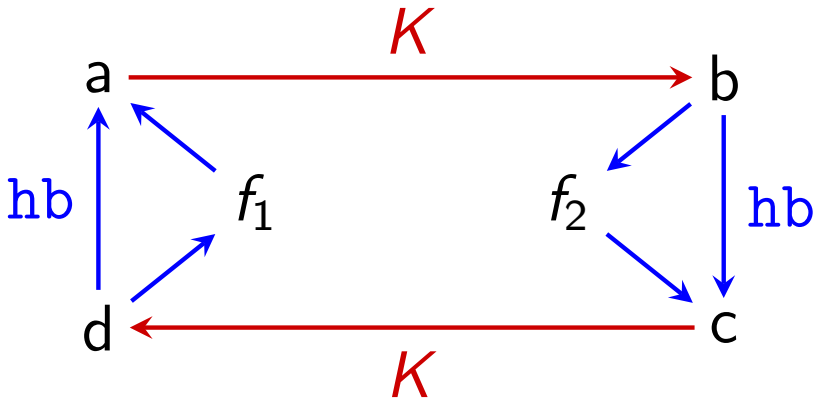
Proof of the simple reduction theorem (1/2)

Recall:

- Recall SC-consistency : $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic.
- Let $\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{rf} \cup \mathtt{sc})^+$ and $K \triangleq (\mathtt{mo} \cup \mathtt{rb}) \setminus \mathtt{hb}$.
- It suffices to prove : $\mathtt{hb} \cup K$ is acyclic.

Consider minimal cycle in $(\mathtt{hb} \cup K)$.

- Cycles with ≤ 1 K-edges disallowed by RA consistency.
- Cycle with two K-edges:



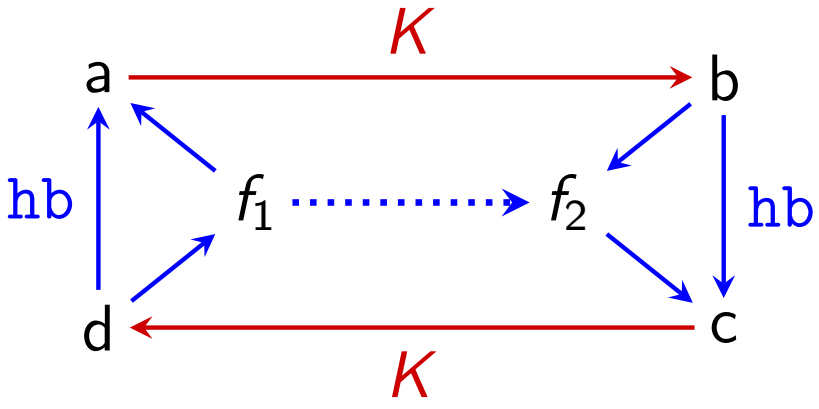
Proof of the simple reduction theorem (1/2)

Recall:

- Recall SC-consistency : $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic.
- Let $\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{rf} \cup \mathtt{sc})^+$ and $K \triangleq (\mathtt{mo} \cup \mathtt{rb}) \setminus \mathtt{hb}$.
- It suffices to prove : $\mathtt{hb} \cup K$ is acyclic.

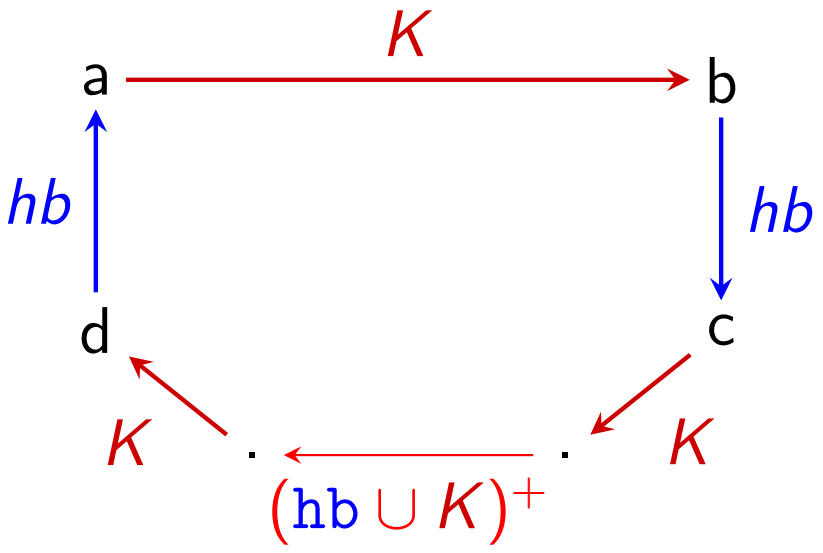
Consider minimal cycle in $(\mathtt{hb} \cup K)$.

- Cycles with ≤ 1 K-edges disallowed by RA consistency.
- Cycle with two K-edges:



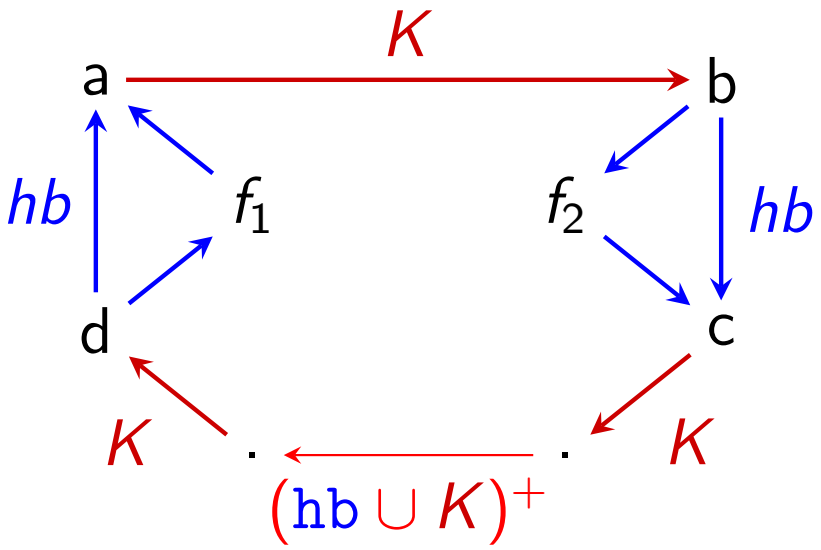
Proof of the simple reduction theorem (2/2)

Finally, consider a cycle with three or more K-edges.



Proof of the simple reduction theorem (2/2)

Finally, consider a cycle with three or more K-edges.



Proof of the simple reduction theorem (2/2)

Finally, consider a cycle with three or more K-edges.