Introduction to the C/C++11 concurrency model

Graduate seminar on weakly consistent concurrency

Ori Lahav

March 14, 2018

The C11 memory model

- Introduced by the ISO C/C++ 2011 standards.
- Defines the semantics of concurrent memory accesses.
- Platform independent.
- Supported by popular compilers (GCC, LLVM, ...).

We will introduce a *simplified* version of (a fragment of) the C11 model as a *prototypical example* of a real world weak concurrency semantics.

Atomics

Two types of accesses

Ordinary (Non-Atomic)

Races are errors

Atomic

Welcome to the expert mode

Atomics

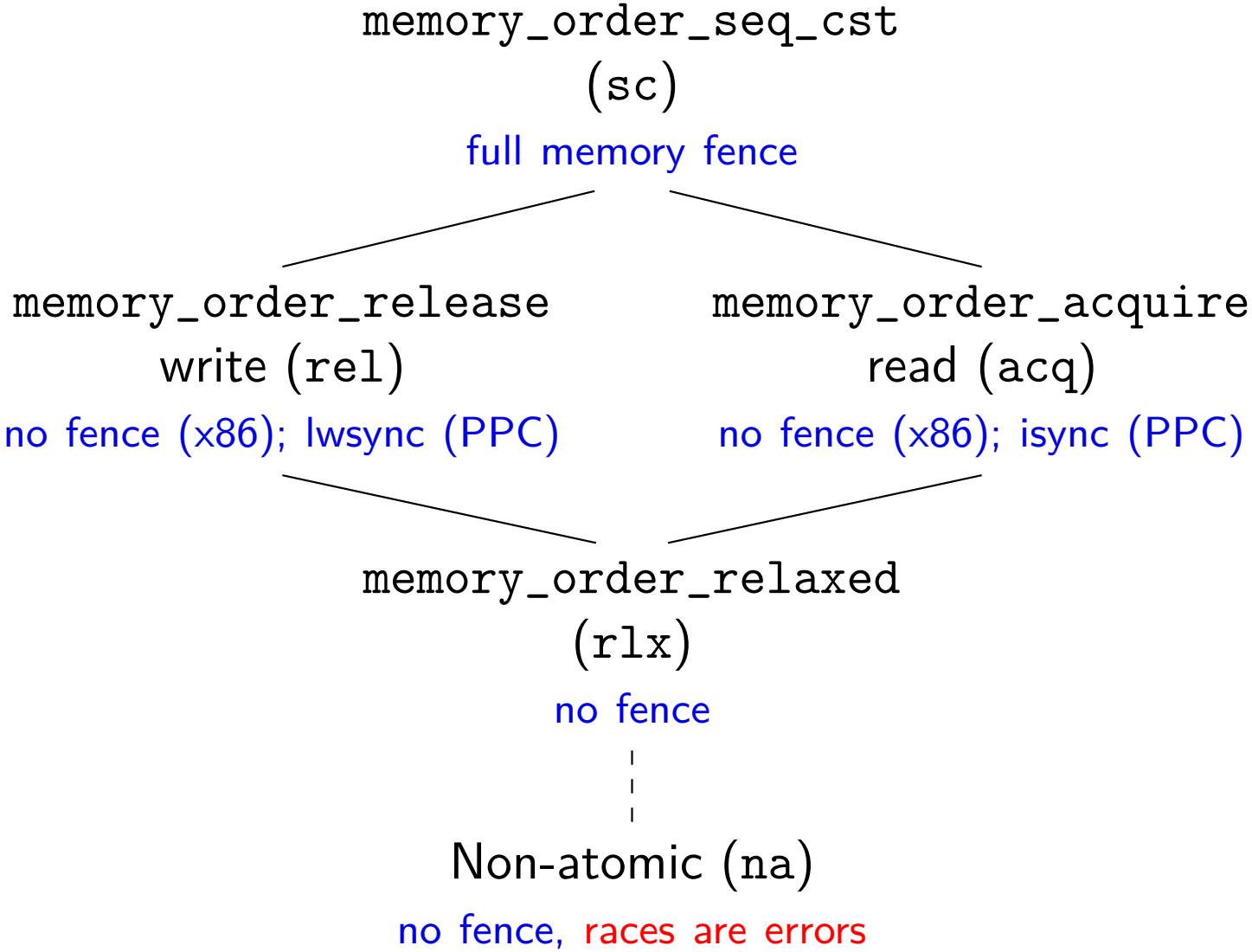
Two types of accesses

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

DRF (data race freedom) guarantee

no data races under SC $\Longrightarrow$ only SC behaviors

A spectrum of access modes



+ Explicit primitives for fences

C11: a declarative memory model

Declarative semantics abstracts away from implementation details.

1. a program $\rightsquigarrow$ a set of directed graphs (called: *execution graphs*)
2. The memory model defines what executions are *consistent*.
3. The semantics of a program is the set of its consistent executions.

Exception: "catch-fire" semantics

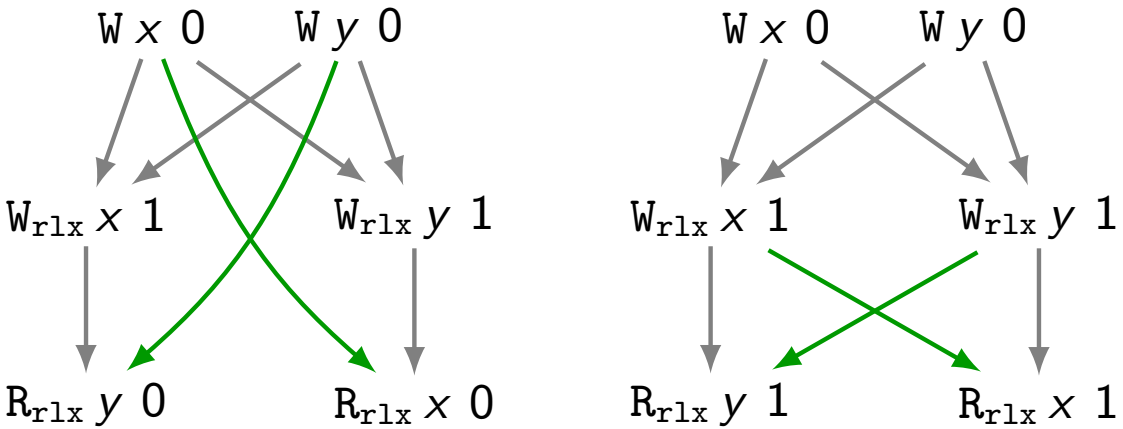
- Existence of at least one "bad" consistent execution (with a forbidden data races) implies undefined behavior.

Execution graphs

Store buffering (SB)

$$x = y = 0$$

$$\begin{array}{l|l} x :=_{\texttt{rlx}} 1 & y :=_{\texttt{rlx}} 1 \\ a := y_{\texttt{rlx}} & b := x_{\texttt{rlx}} \end{array}$$



Relations

- Program order, po
- Reads-from, rf (we require that *every read reads from some write*).

C/C++11 formal model

[Vafeiadis & Narayan OOPSLA'13]

$$
\begin{array}{rcl}
[\![-]\!] & : & \mathsf{CExp} \to \mathbb{P}(\langle \mathit{res} : \mathsf{Val} \cup \{\bot\}, \mathcal{A} : \mathbb{P}(\mathsf{AName}), \mathsf{lab} : \mathcal{A} \to \mathsf{Act}, \mathsf{sb} : \mathbb{P}(\mathcal{A} \times \mathcal{A}), \mathit{fst} : \mathcal{A}, \mathit{lst} : \mathcal{A}\rangle) \\
[\![v]\!] & \stackrel{\text{def}}{=} & \{\langle v, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathsf{skip}\} \\
[\![\mathbf{alloc}()]\!] & \stackrel{\text{def}}{=} & \{\langle \ell, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \ell \in \mathsf{Loc} \wedge \mathsf{lab}(a) = \mathrm{A}(\ell)\} \\
[\![[v]_Z := v']\!] & \stackrel{\text{def}}{=} & \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{W}_Z(v, v')\} \\
[\![[v]_Z]\!] & \stackrel{\text{def}}{=} & \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge v' \in \mathsf{Val} \wedge \mathsf{lab}(a) = \mathrm{R}_Z(v, v')\} \\
[\![\mathbf{CAS}_{X,Y}(v, v_o, v_n)]\!] & \stackrel{\text{def}}{=} & \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge v' \in \mathsf{Val} \wedge v' \neq v_o \wedge \mathsf{lab}(a) = \mathrm{R}_Y(v, v')\} \\
 & & \cup\ \{\langle v_o, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{RMW}_X(v, v_o, v_n)\} \\
[\![\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2]\!] & \stackrel{\text{def}}{=} & \{\langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \mid \langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \in [\![E_1]\!]\} \\
 & & \cup\ \{\langle \mathit{res}_2, \mathcal{A}_1 \uplus \mathcal{A}_2, \mathsf{lab}_1 \cup \mathsf{lab}_2, \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(\mathit{lst}_1, \mathit{fst}_2)\}, \mathit{fst}_1, \mathit{lst}_2\rangle \mid \\
 & & \quad \langle v_1, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \in [\![E_1]\!] \wedge \langle \mathit{res}_2, \mathcal{A}_2, \mathsf{lab}_2, \mathsf{sb}_2, \mathit{fst}_2, \mathit{lst}_2\rangle \in [\![E_2[v_1/x]]\!]\} \\
[\![\mathbf{repeat}\ E\ \mathbf{end}]\!] & \stackrel{\text{def}}{=} & \{\langle \mathit{res}_N, \biguplus_{i\in[1..N]} \mathcal{A}_i, \bigcup_{i\in[1..N]} \mathsf{lab}_i, \bigcup_{i\in[1..N]} \mathsf{sb}_i \cup \{(\mathit{lst}_1, \mathit{fst}_2), \ldots, (\mathit{lst}_{N-1}, \mathit{fst}_N)\}, \mathit{fst}_1, \mathit{lst}_N\rangle \mid \\
 & & \quad \forall i.\ \langle \mathit{res}_i, \mathcal{A}_i, \mathsf{lab}_i, \mathsf{sb}_i, \mathit{fst}_i, \mathit{lst}_i\rangle \in [\![E]\!] \wedge (i \neq N \implies \mathit{res}_i = 0) \wedge \mathit{res}_N \neq 0\} \\
[\![E_1 \| E_2]\!] & \stackrel{\text{def}}{=} & \{\langle \mathsf{combine}(\mathit{res}_1, \mathit{res}_2), \mathcal{A}_1 \uplus \mathcal{A}_2 \uplus \{a_{\text{fork}}, a_{\text{join}}\}, \mathsf{lab}_1 \cup \mathsf{lab}_2 \cup \{a_{\text{fork}} \mapsto \mathsf{skip}, a_{\text{join}} \mapsto \mathsf{skip}\}, \\
 & & \quad \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(a_{\text{fork}}, \mathit{fst}_1), (a_{\text{fork}}, \mathit{fst}_2), (\mathit{lst}_1, a_{\text{join}}), (\mathit{lst}_2, a_{\text{join}})\}, a_{\text{fork}}, a_{\text{join}}\rangle \mid \\
 & & \quad \langle \mathit{res}_1, \mathcal{A}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \in [\![E_1]\!] \wedge \langle \mathit{res}_2, \mathcal{A}_2, \mathsf{sb}_2, \mathit{fst}_2, \mathit{lst}_2\rangle \in [\![E_2]\!] \wedge a_{\text{fork}}, a_{\text{join}} \in \mathsf{AName}\}
\end{array}
$$

Figure 2. Semantics of closed program expressions.

$$
\begin{array}{rr}
\nexists x.\ \mathsf{hb}(x,x) & \text{(IrreflexiveHB)} \\
\forall \ell.\ \mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{iswrite}_\ell(a)\}, \mathsf{mo}) \wedge \mathsf{hb}_\ell \subseteq \mathsf{mo} & \text{(ConsistentMO)} \\
\mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{isSeqCst}(a)\}, \mathsf{sc}) \wedge \mathsf{hb}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} \wedge \mathsf{mo}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} & \text{(ConsistentSC)} \\
\forall b.\ \mathsf{rf}(b) \neq \bot \iff \exists \ell, a.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{isread}_\ell(b) \wedge \mathsf{hb}(a,b) & \text{(ConsistentRFdom)} \\
\forall a, b.\ \mathsf{rf}(b) = a \implies \exists \ell, v.\ \mathsf{iswrite}_{\ell,v}(a) \wedge \mathsf{isread}_{\ell,v}(b) \wedge \neg\mathsf{hb}(b,a) & \text{(ConsistentRF)} \\
\forall a, b.\ \mathsf{rf}(b) = a \wedge (\mathsf{mode}(a) = \mathrm{na} \vee \mathsf{mode}(b) = \mathrm{na}) \implies \mathsf{hb}(a,b) & \text{(ConsistentRFna)} \\
\forall a, b.\ \mathsf{rf}(b) = a \wedge \mathsf{isSeqCst}(b) \implies \mathsf{isc}(a,b) \vee \neg\mathsf{isSeqCst}(a) \wedge (\forall x.\ \mathsf{isc}(x,b) \Rightarrow \neg\mathsf{hb}(a,x)) & \text{(RestrSCReads)} \\
\nexists a, b.\ \mathsf{hb}(a,b) \wedge \mathsf{mo}(\mathsf{rf}(b), \mathsf{rf}(a)) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) & \text{(CoherentRR)} \\
\nexists a, b.\ \mathsf{hb}(a,b) \wedge \mathsf{mo}(\mathsf{rf}(b), a) \wedge \mathsf{iswrite}(a) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) & \text{(CoherentWR)} \\
\nexists a, b.\ \mathsf{hb}(a,b) \wedge \mathsf{mo}(b, \mathsf{rf}(a)) \wedge \mathsf{iswrite}(b) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) & \text{(CoherentRW)} \\
\forall a.\ \mathsf{isrmw}(a) \wedge \mathsf{rf}(a) \neq \bot \implies \mathsf{mo}(\mathsf{rf}(a), a) \wedge \nexists c.\ \mathsf{mo}(\mathsf{rf}(a), c) \wedge \mathsf{mo}(c, a) & \text{(AtomicRMW)} \\
\forall a, b, \ell.\ \mathsf{lab}(a) = \mathsf{lab}(b) = \mathrm{A}(\ell) \implies a = b & \text{(ConsistentAlloc)}
\end{array}
$$

$$
\begin{array}{rcl}
\text{where } \mathsf{iswrite}_{\ell,v}(a) & \stackrel{\text{def}}{=} & \exists X, v_{\text{old}}.\ \mathsf{lab}(a) \in \{\mathrm{W}_X(\ell, v), \mathrm{RMW}_X(\ell, v_{\text{old}}, v)\} \qquad \mathsf{iswrite}_\ell(a) \stackrel{\text{def}}{=} \exists v.\ \mathsf{iswrite}_{\ell,v}(a) \\
\mathsf{isread}_{\ell,v}(a) & \stackrel{\text{def}}{=} & \exists X, v_{\text{new}}.\ \mathsf{lab}(a) \in \{\mathrm{R}_X(\ell, v), \mathrm{RMW}_X(\ell, v, v_{\text{new}})\} \qquad \text{etc.} \\
\mathsf{rsElem}(a,b) & \stackrel{\text{def}}{=} & \mathsf{sameThread}(a,b) \vee \mathsf{isrmw}(b) \\
\mathsf{rseq}(a) & \stackrel{\text{def}}{=} & \{a\} \cup \{b \mid \mathsf{rsElem}(a,b) \wedge \mathsf{mo}(a,b) \wedge (\forall c.\ \mathsf{mo}(a,c) \wedge \mathsf{mo}(c,b) \Rightarrow \mathsf{rsElem}(a,c))\} \\
\mathsf{sw} & \stackrel{\text{def}}{=} & \{(a,b) \mid \mathsf{mode}(a) \in \{\mathrm{rel}, \mathrm{rel\_acq}, \mathrm{sc}\} \wedge \mathsf{mode}(b) \in \{\mathrm{acq}, \mathrm{rel\_acq}, \mathrm{sc}\} \wedge \mathsf{rf}(b) \in \mathsf{rseq}(a)\} \\
\mathsf{hb} & \stackrel{\text{def}}{=} & (\mathsf{sb} \cup \mathsf{sw})^+ \\
\mathsf{hb}_\ell & \stackrel{\text{def}}{=} & \{(a,b) \in \mathsf{hb} \mid \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b)\} \\
X_{\mathsf{SeqCst}} & \stackrel{\text{def}}{=} & \{(a,b) \in X \mid \mathsf{isSeqCst}(a) \wedge \mathsf{isSeqCst}(b)\} \\
\mathsf{isc}(a,b) & \stackrel{\text{def}}{=} & \mathsf{iswrite}_{\mathsf{locs}(b)}(a) \wedge \mathsf{sc}(a,b) \wedge \nexists c.\ \mathsf{sc}(a,c) \wedge \mathsf{sc}(c,b) \wedge \mathsf{iswrite}_{\mathsf{locs}(b)}(c)
\end{array}
$$

Figure 3. Axioms satisfied by consistent C11 executions, $\mathsf{Consistent}(\mathcal{A}, \mathsf{lab}, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc})$.

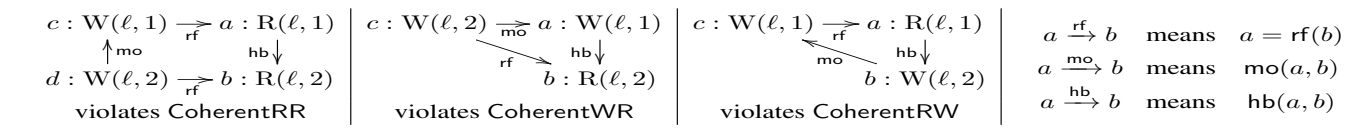
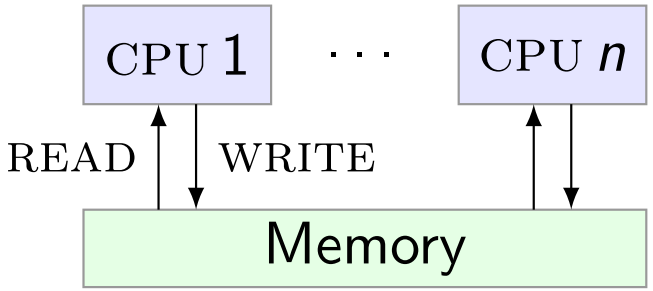


Figure 4. Sample executions violating coherency conditions (Batty et al. 2011).

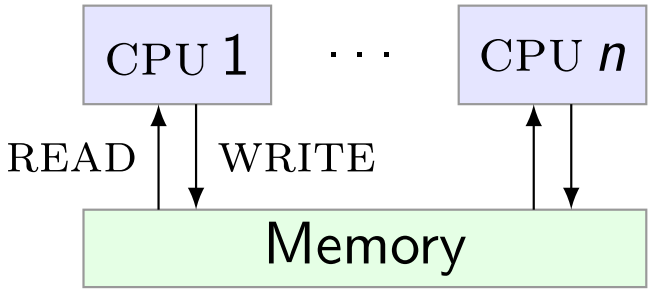
Sequential consistency

Before C/C++11, how would you define SC as a declarative memory model?



Sequential consistency

Before C/C++11, how would you define SC as a declarative memory model?



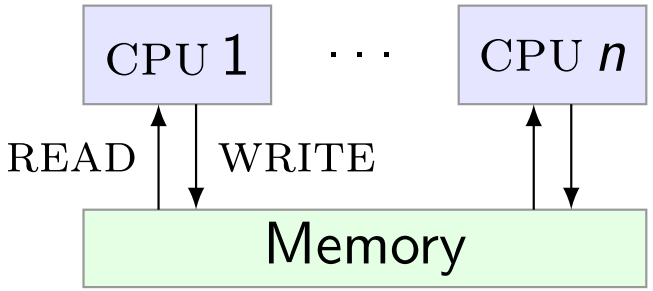
Definition (Lamport '79)

G is SC-*consistent* if there exists a relation $\texttt{sc}$ such that:

- $\texttt{sc}$ is a total order on the events of G.
- $\texttt{po} \cup \texttt{rf} \subseteq \texttt{sc}$.
- If $\langle a, b\rangle \in \texttt{rf}$ then there does not exist $c \in \mathtt{W}_{\mathtt{loc}(a)}$ such that $\langle a, c\rangle \in \texttt{sc}$ and $\langle c, b\rangle \in \texttt{sc}$.

Sequential consistency

Before C/C++11, how would you define SC as a declarative memory model?



Definition (Lamport '79)

G is SC-*consistent* if there exists a relation $\mathtt{sc}$ such that:

- $\mathtt{sc}$ is a total order on the events of G.
- $\mathtt{po} \cup \mathtt{rf} \subseteq \mathtt{sc}$.
- If $\langle a, b\rangle \in \mathtt{rf}$ then there does not exist $c \in \mathtt{W}_{\mathtt{loc}(a)}$ such that $\langle a, c\rangle \in \mathtt{sc}$ and $\langle c, b\rangle \in \mathtt{sc}$.

- We will later see alternative more "economical" definitions that do not require a total order on all events of G.

Ingredients of execution graph consistency in C/C++11

1. SC-per-location (a.k.a. coherence)
2. Release/acquire synchronization
3. Global conditions on SC accesses

Ingredients of execution graph consistency in C/C++11

1. SC-per-location (a.k.a. coherence)
2. Release/acquire synchronization
3. Global conditions on SC accesses

SC-per-location

Definition (Declarative definition of SC)

G is *SC-consistent* if there exists a relation $\mathtt{sc}$ such that:

- $\mathtt{sc}$ is a total order on the events of G.
- $\mathtt{po} \cup \mathtt{rf} \subseteq \mathtt{sc}$.
- If $\langle a, b\rangle \in \mathtt{rf}$ then there does not exist $c \in \mathtt{W}_{\mathtt{loc}(a)}$ such that $\langle a, c\rangle \in \mathtt{sc}$ and $\langle c, b\rangle \in \mathtt{sc}$.

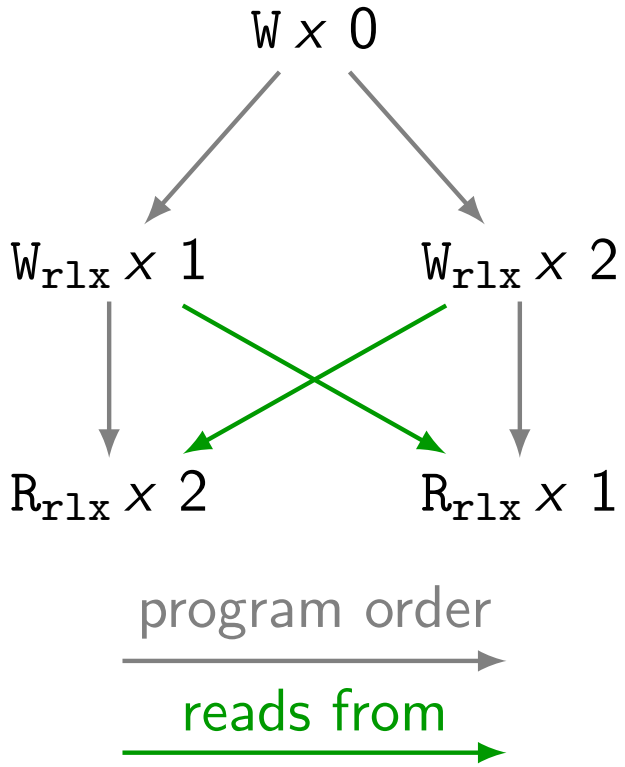
Definition (SC-per-location)

G satisfies *SC-per-location* if for every location x, there exists a relation $\mathtt{sc}_x$ such that:

- $\mathtt{sc}_x$ is a total order on the events of G that access x.
- $(\mathtt{po} \cup \mathtt{rf}) \cap \{\langle a, b\rangle \mid \mathtt{loc}(a) = \mathtt{loc}(b) = x\} \subseteq \mathtt{sc}_x$.
- If $\langle a, b\rangle \in \mathtt{rf}$ then there does not exist $c \in \mathtt{W}_x$ such that $\langle a, c\rangle \in \mathtt{sc}_x$ and $\langle c, b\rangle \in \mathtt{sc}_x$.

SC-per-location: Example

$$
\begin{array}{c}
x = 0 \\
x :=_{\texttt{rlx}} 1 \;\Big\|\; x :=_{\texttt{rlx}} 2 \\
a := x_{\texttt{rlx}} \;\Big\|\; b := x_{\texttt{rlx}}
\end{array}
$$



inconsistent!

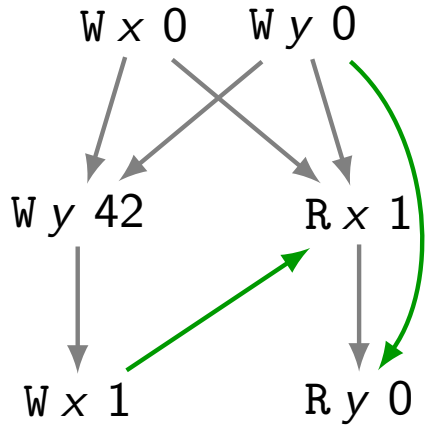
Release/acquire synchronization

SC-per-location is often too weak:

- It does not support the message passing idiom:

Message passing (MP)

$y := 42;$	$\Vert$	$a := x;$ // 1
$x := 1;$	$\Vert$	$b := y;$ // 0



Implementing locks?

Can we implement a lock under SC-per-location?

Implementing locks?

Can we implement a lock under SC-per-location?

Spinlock implementation

lock(l) :

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

unlock(l) :

$l := 0$

Implementing locks?

Can we implement a lock under SC-per-location?

Spinlock implementation

lock(l) :

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

unlock(l) :

$l := 0$

Lock example

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

Under SC-per-location, the spinlock implementation does not guarantee mutual exclusion.

Synchronization in C/C++11 through examples

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

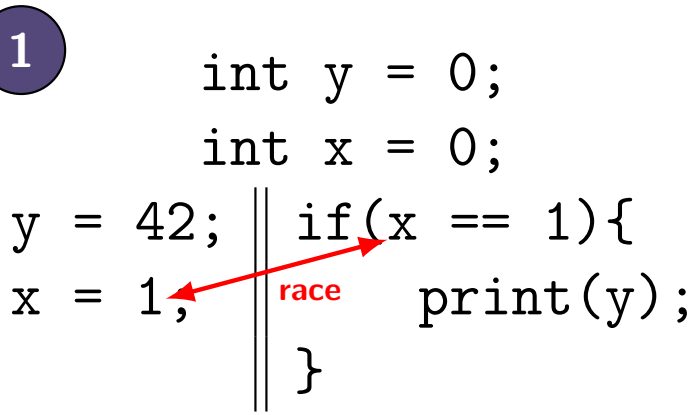
Synchronization in C/C++11 through examples

1

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

Synchronization in C/C++11 through examples

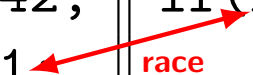
1



Synchronization in C/C++11 through examples

1

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



2

```
        int y = 0;
    atomic<int> x = 0;
y = 42;   ‖ if(x_rlx == 1){
x =_rlx 1; ‖     print(y);
          ‖ }
```

Synchronization in C/C++11 through examples

1

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

race

2

```
int y = 0;
atomic<int> x = 0;
y = 42;     ||  if(x_rlx == 1){
x =_rlx 1;  ||    print(y);
            ||  }
```

race

Synchronization in C/C++11 through examples

1

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

race

2

```
        int y = 0;
    atomic<int> x = 0;
y = 42;    || if(x_rlx == 1){
x =_rlx 1; ||    print(y);
           || }
```

race

3

```
        int y = 0;
    atomic<int> x = 0;
y = 42;    || if(x_acq == 1){
x =_rel 1; ||    print(y);
           || }
```

Synchronization in C/C++11 through examples

1

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


2

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
x =_rlx 1;       ||      print(y);
                 ||  }
```


3

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_acq == 1){
x =_rel 1;       ||      print(y);
                 ||  }
```

Synchronization in C/C++11 through examples

1

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

race

2

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
x =_rlx 1;       ||      print(y);
                 ||  }
```

race

3

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_acq == 1){
x =_rel 1;       ||      print(y);
                 ||  }
```

rf

sw

Synchronization in C/C++11 through examples

1

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


2

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
x =_rlx 1;       ||      print(y);
                 ||  }
```


3

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_acq == 1){
x =_rel 1;       ||      print(y);
                 ||  }
```


4

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
fence_rel;       ||      fence_acq;
x =_rlx 1;       ||      print(y);
                 ||  }
```

Synchronization in C/C++11 through examples

1

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

race

2

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
x =_rlx 1;       ||    print(y);
                 ||  }
```

race

3

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_acq == 1){
x =_rel 1;       ||    print(y);
                 ||  }
```

rf

sw

4

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
fence_rel;       ||    fence_acq;
x =_rlx 1;       ||    print(y);
                 ||  }
```

rf

Synchronization in C/C++11 through examples

1

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

race

2

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
x =_rlx 1;       ||    print(y);
                 ||  }
```

race

3

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_acq == 1){
x =_rel 1;       ||    print(y);
                 ||  }
```

rf, sw

4

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
fence_rel;       ||    fence_acq;
x =_rlx 1;       ||    print(y);
                 ||  }
```

rf, sw

The “happens-before” relation

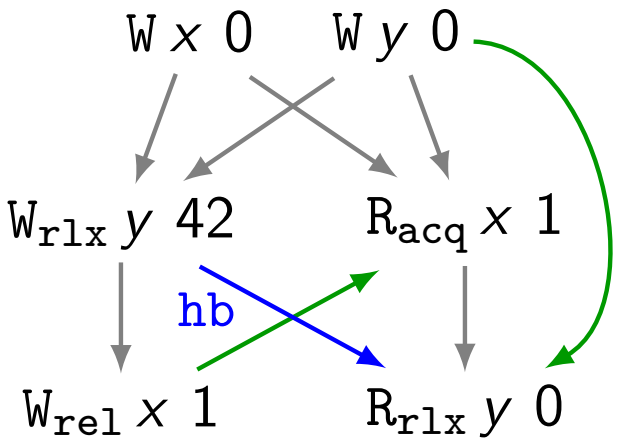
Definition (happens-before)

$$\frac{a \xrightarrow{\text{po}} b}{a \xrightarrow{\text{hb}} b}$$

a: `W Rel` b: `R Acq`

$$\frac{a \xrightarrow{\text{rf}} b}{a \xrightarrow{\text{hb}} b}$$

$$\frac{a \xrightarrow{\text{hb}} b \qquad b \xrightarrow{\text{hb}} c}{a \xrightarrow{\text{hb}} c}$$



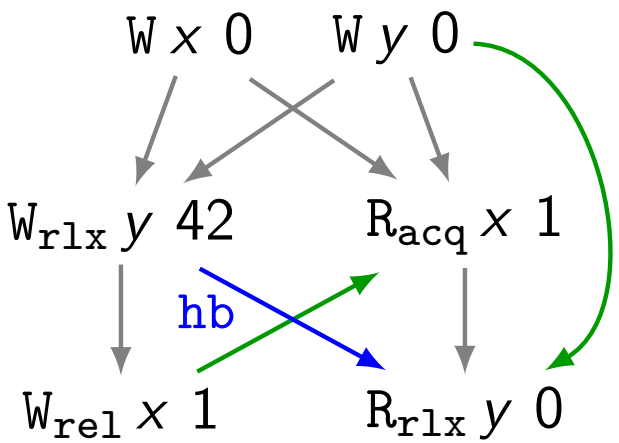
C11 coherence

- Require (for every location x) that :

$$\mathtt{hb} \cap \{\langle a, b\rangle \mid \mathtt{loc}(a) = \mathtt{loc}(b) = x\} \subseteq \mathtt{sc}_x$$

(instead of $(\mathtt{po} \cup \mathtt{rf}) \cap \{\langle a, b\rangle \mid \mathtt{loc}(a) = \mathtt{loc}(b) = x\} \subseteq \mathtt{sc}_x$).

- Using acquire CAS's and release writes, we can implement locks.



Races

Using $\texttt{hb}$, we can formally define a race in this model.

Definition

Two events a, b form a *race* in an execution G if the following hold:

- $\texttt{loc}(a) = \texttt{loc}(b)$.
- $a \in \texttt{W}$ or $b \in \texttt{W}$ ($\texttt{W}$ denotes the set of writes and RMWs).
- $\langle a, b \rangle \notin \texttt{hb}$ and $\langle b, a \rangle \notin \texttt{hb}$.

SC accesses and fences

Store buffer

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

How to guarantee only SC behaviors (*i.e.*, $a = 1 \vee b = 1$)?

SC accesses and fences

Store buffer

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

How to guarantee only SC behaviors (*i.e.*, $a = 1 \vee b = 1$)?

$$\begin{array}{l} x :=_{\mathtt{sc}} 1; \\ a := y_{\mathtt{sc}}; \end{array} \ \Big\| \ \begin{array}{l} y :=_{\mathtt{sc}} 1; \\ b := x_{\mathtt{sc}}; \end{array} \quad \cong \quad \begin{array}{l} x :=_{\mathtt{rlx}} 1; \\ \mathbf{fence}_{\mathtt{sc}}; \\ a := y_{\mathtt{rlx}}; \end{array} \ \Big\| \ \begin{array}{l} y :=_{\mathtt{rlx}} 1; \\ \mathbf{fence}_{\mathtt{sc}}; \\ b := x_{\mathtt{rlx}}; \end{array}$$

Semantics of SC accesses

Basic idea: Require an $\mathtt{sc}$-order on SC accesses:

- $\mathtt{sc}$ is a total order on all SC accesses in G.
- $\mathtt{hb} \cap \{\langle a, b\rangle \mid \mathtt{mod}(a) = \mathtt{mod}(b) = \mathtt{sc}\} \subseteq \mathtt{sc}$.
- If $\langle a, b\rangle \in \mathtt{rf}$ then there does not exist $c \in \mathtt{W}_{\mathtt{loc}(a)}$ such that $\langle a, c\rangle \in \mathtt{sc}$ and $\langle c, b\rangle \in \mathtt{sc}$.

- Semantics of SC fences?
- Accesses to the same location in both SC and non-SC modes?

This is a complicated part of the model:
[Repairing Sequential Consistency in C/C++11 PLDI'17]

Revisiting sequential consistency

Definition

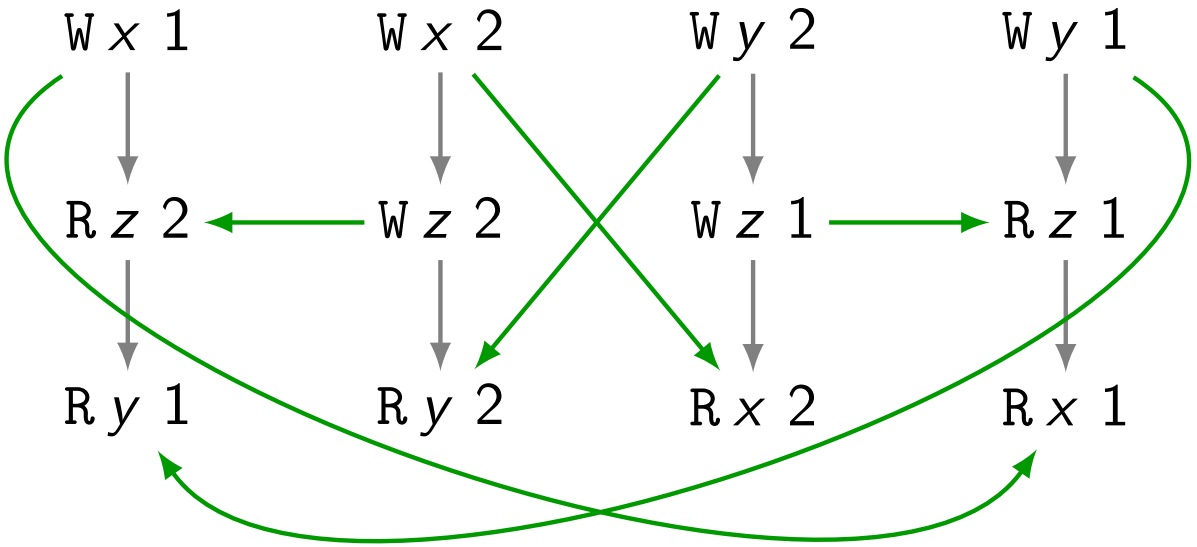
G is SC-*consistent* if there exists a relation $\texttt{sc}$ such that:

- $\texttt{sc}$ is a total order on the events of G.
- $\texttt{po} \cup \texttt{rf} \subseteq \texttt{sc}$.
- If $\langle a, b \rangle \in \texttt{rf}$ then there does not exist $c \in \texttt{W}_{\texttt{loc}(a)}$ such that $\langle a, c \rangle \in \texttt{sc}$ and $\langle c, b \rangle \in \texttt{sc}$.

Can we avoid ordering *all* events?

Example

[Sezgin '04]



Alternative definition of SC

Definition (Modification order (a.k.a. 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 $\mathtt{W}_x$.

Definition (Alternative SC definition)

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

- There exists a modification order $\mathtt{mo}$ for G such that $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{fr}$ is acyclic where:
  - $\mathtt{fr} \stackrel{\text{def}}{=} \mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathit{id}$ (from-reads / reads-before)

Equivalence

Theorem

The two SC definitions are equivalent.

Proof (sketch).

Original SC $\implies$ alternative SC:

- Take $\texttt{mo}_x$ to be the restriction of $\texttt{sc}$ on $\texttt{W}_x$.
- Then, $\texttt{po} \cup \texttt{rf} \cup \texttt{mo} \cup \texttt{fr} \subseteq \texttt{sc}$.

Alternative SC $\implies$ original SC:

- Take $\texttt{sc}$ to be any total order extending $\texttt{po} \cup \texttt{rf} \cup \texttt{mo} \cup \texttt{fr}$. □

Alternative definition of SC-per-location

Definition (SC-per-location)

G satisfies *SC-per-location* if for every location x, there exists a relation $\mathtt{sc}_x$ such that:

- $\mathtt{sc}_x$ is a total order on the events of G that access x.
- $(\mathtt{po} \cup \mathtt{rf}) \cap \{\langle a, b\rangle \mid \mathtt{loc}(a) = \mathtt{loc}(b) = x\} \subseteq \mathtt{sc}_x$.
- If $\langle a, b\rangle \in \mathtt{rf}$ then there does not exist $c \in \mathtt{W}_x$ such that $\langle a, c\rangle \in \mathtt{sc}_x$ and $\langle c, b\rangle \in \mathtt{sc}_x$.

Definition (Alternative SC-per-location definition)

An execution graph G is satisfies *SC-per-location* if the following holds:

- There exists a modification order $\mathtt{mo}$ for G such that $\mathtt{po}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{fr}$ is acyclic where:
  - $\mathtt{fr} \stackrel{\text{def}}{=} \mathtt{rf}^{-1}; \mathtt{mo} \setminus id$ (from-reads / reads-before)
  - $\mathtt{po}|_{\mathtt{loc}} = \mathtt{po} \cap (\langle a, b\rangle \mid \mathtt{loc}(a) = \mathtt{loc}(b))$

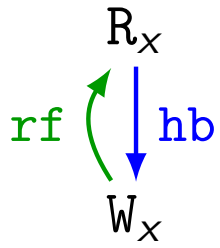
Back to C11

- Similar "optimization" is done in the full C11 model, requiring that:

$$\texttt{hb}|_{\texttt{loc}} \cup \texttt{rf} \cup \texttt{mo} \cup \texttt{fr} \text{ is acyclic}$$

- This acyclicity condition can be simplified to several "bad patterns".

“Bad patterns” I



$a := x \;/\!/\; 1$
$x := 1$

no-future-read

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

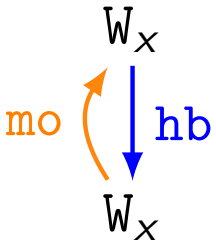
rmw-1

Notations:

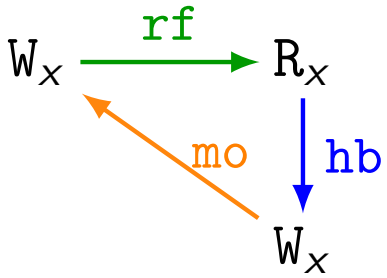
- $\mathtt{W}$ is either a write or an RMW.
- $\mathtt{R}$ is either a read or an RMW.

“Bad patterns” II

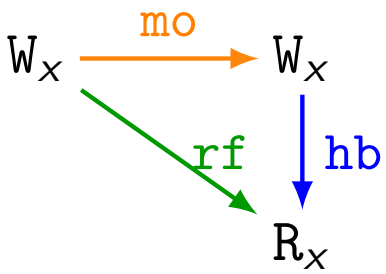
$$\begin{array}{l|l} x := 1 & a := x \mathbin{/\!/} 2 \\ x := 2 & a := x \mathbin{/\!/} 1 \end{array}$$



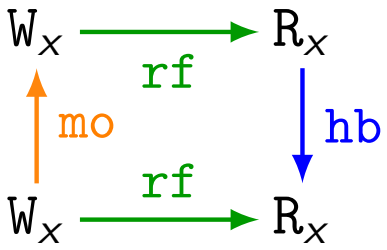
coherence-ww



coherence-rw

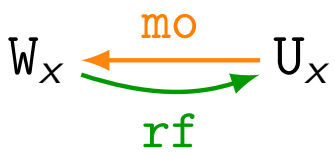


coherence-wr

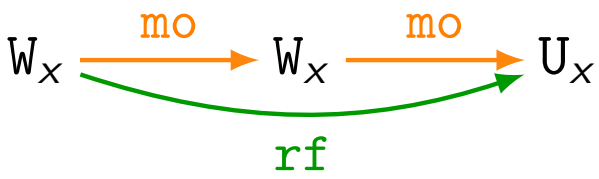


coherence-rr

“Bad patterns” III



rmw-2



atomicity

Alternative definition via “Bad patterns”

Theorem

Let mo *be a modification order for an execution graph* G*.*
$\mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{fr}$ *is acyclic iff the following hold:*

- rf;hb *is irreflexive.* *(no-future-read)*
- mo;hb *is irreflexive.* *(coherence-ww)*
- mo;rf;hb *is irreflexive.* *(coherence-rw)*
- fr;hb *is irreflexive.* *(coherence-wr)*
- fr;rf;hb *is irreflexive.* *(coherence-rr)*
- rf *is irreflexive.* *(rmw-1)*
- mo;rf *is irreflexive.* *(rmw-2)*
- fr;mo *is irreflexive.* *(rmw-atomicity)*

Side note: Java plain accesses do not guarantee (coherence-rr).

Takeaways

- C/C++11 concurrency semantics is specified *declaratively* using formal constrains on execution graphs.
- This is a *highly flexible* framework that abstracts away all execution details.
- Reasoning about program behaviors requires analyzing *possible cycles* in execution graphs.
- A declarative approach to concurrency semantics is prominent in other systems as well:
  - *Multicore architectures:*
    Jade Alglave, Luc Maranget, Michael Tautschnig.
    Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory.
  - *Distributed systems:*
    Sebastian Burckhardt.
    Principles of Eventual Consistency.