The C/C++11 memory model

Ori Lahav Viktor Vafeiadis

30 August 2017

Recap

We considered a simplified C11 model:

- Each memory accesses has a *mode*:
  - Reads: **rlx** or **acq**
  - Writes: **rlx** or **rel**
  - RMWs: **rlx**, **acq**, **rel** or **acq-rel**
- Synchronization:

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

- Happens-before:

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

- C11-consistent wrt $\mathtt{mo}$:

$$\mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb} \text{ is acyclic}$$

- C11-consistent:

$$\text{complete \& C11-consistent wrt some } \mathtt{mo}$$

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

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);
           || }
```

race

3

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

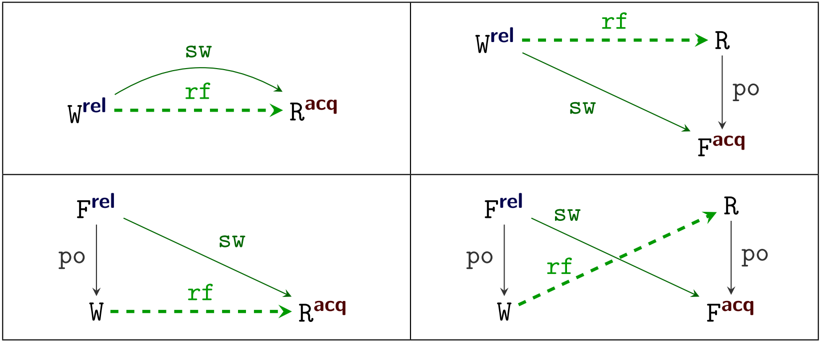
rf, sw

4

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

rf, sw

The “synchronizes-with” relation

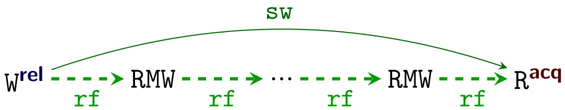


$$\mathtt{sw} \triangleq ([\mathtt{W}^{\sqsupseteq \mathsf{rel}}] \cup [\mathtt{F}^{\sqsupseteq \mathsf{rel}}]; \mathtt{po}); \mathtt{rf}; ([\mathtt{R}^{\sqsupseteq \mathsf{acq}}] \cup \mathtt{po}; [\mathtt{F}^{\sqsupseteq \mathsf{acq}}])$$

Fence modes

Release sequences (RMW's)

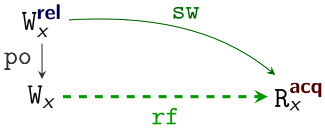
$x_{\mathsf{rlx}} := 42;$ $y_{\mathsf{rel}} := 1$	$a := \mathbf{FAI}_{\mathsf{rlx}}(y);$ // 1	$b := y_{\mathsf{acq}};$ // 2 $c := x_{\mathsf{rlx}};$ // 0



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

Release sequences (thread internal)

$$
\begin{array}{l|l}
x_{\mathbf{rlx}} := 42; & \\
y_{\mathbf{rel}} := 1; & a := y_{\mathbf{acq}} \quad /\!/ \ 2 \\
y_{\mathbf{rlx}} := 2; & b := x_{\mathbf{rlx}} \quad /\!/ \ 0
\end{array}
$$



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

C11 "synchronizes-with" relation

Read modes

na → **rlx** → **acq** → **sc**

Write modes

na → **rlx** → **rel** → **sc**

RMW modes

rlx → **acq** → **acq-rel**
rlx → **acq-rel** → **sc**
rlx → **rel** → **acq-rel**

Fence modes

acq → **acq-rel** → **sc**
rel → **acq-rel**

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

$$\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{sw})^{+}$$

“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:

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

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

Definition (Allowed outcome under C11)

An outcome O is *allowed* for a program P under C11 if there exists an execution graph G such that:

- G is an execution graph of P
- G is C11-consistent.
- G has outcome O or G is C11-racy.

Definition

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

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

Definition

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

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

- The most involved part of the model, due to the possible mixing of different access modes to the same location.
- Currently (August 2017) under revision.
- *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

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

$$\texttt{psc}_{\texttt{F}} \triangleq [\texttt{F}^{\textsf{sc}}]; (\texttt{hb} \cup \texttt{hb}; \texttt{eco}; \texttt{hb}); [\texttt{F}^{\textsf{sc}}] \quad \text{(partial SC order on fences)}$$

Condition on SC fences

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

Example: SB with fences

$$x = y = 0$$

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

✗ behavior disallowed

Exercise: ARC

$$a = \texttt{new}(v)$$
$$y = \texttt{read}(a)$$
$$\texttt{clone}(a)$$
$$\texttt{drop}(a)$$

```
new(v){
  a = alloc();
  a.data = v;
  a.count = 1;
  return a;
}

read(a){
  return a.data;
}
```

```
clone(a){
  FADD(a.count, +1);
}

drop(a){
  t = FADD(a.count, -1);
  if(t == 1){
    free(a);
  }
}
```

FADD = fetch_and_add

Exercise: seqlock

```
writer(v1,v2) {
  local a,b;
  do {
    a = s;
    if (a % 2 == 1)
      continue;
    b = CAS(s,a,a+1);
  } while (¬b);
  x1 = v1;
  x2 = v2;
  s = a + 2;
}
```

```
reader(t1,t2) {
  local a,b;
  while (1) {
    a = s;
    if (a % 2 == 1)
      continue;
    t1 = x1;
    t2 = x2;
    b = s;
    if (a==b) return;
  }
}
```