Principles of Concurrency

Lecture 12
Memory Models: C11

Based on:
- Foundations of the C++ Concurrency Memory Model, Boehm and Adve, PLDI’08
- Mathematizing C++ Concurrency, Batty et. al, POPL’11
- Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it (Vafeiadis et. al, POPL’15)
Principles

- Most language definitions do not define a memory model
  - E.g., Rust, Python, Haskell, etc.
- But, there are several important exceptions
  - C/C++11
  - Java
- A language memory model provides guidance to developers and compiler writers on expected behaviors

Initially $X = Y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r1 = X$</td>
<td>$r2 = Y$</td>
</tr>
<tr>
<td>if ($r1 == 1$)</td>
<td>if ($r2 == 1$)</td>
</tr>
<tr>
<td>$Y = 1$</td>
<td>$X = 1$</td>
</tr>
</tbody>
</table>

Reasoning about the behavior of this program requires a precise definition on the values a thread is allowed to witness

Is outcome $r1 = r2 = 1$ allowed?

```c
struct s { char a; char b; } x;

Thread 1:    Thread 2:
$x.a = 1;$    $x.b = 1;$

Thread 1 is not equivalent to:
struct s tmp = x;
tmp.a = 1;
x = tmp;
```

Compiler transformations must be aware of threads
C11 Memory Model

- If a program has a sequentially consistent (SC) execution which contains a data race, then its behavior is undefined. The definition has no notion of a benign race.
- Otherwise, the program behaves according to one of its SC executions

Enforcing SC behavior is expensive because it requires that all writes execute atomically with respect to subsequent reads

Initially $X=Y=0$

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X=1$</td>
<td>$Y=1$</td>
<td>$r1=X$</td>
<td>$r3=Y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$fence$</td>
<td>$fence$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$r2=Y$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$r4=X$</td>
</tr>
</tbody>
</table>

$r1=1$, $r2=0$, $r3=1$, $r4=0$ violates write atomicity

If writes execute non-atomically, then they might get propagated to reads out-of-order
Data Races and Compiler Transformations

```c
unsigned i = x;
if (i < 2) {
    foo: ...  
    switch (i) {
        case 0: ...; break;
        case 1: ...; break;
        default: ...;
    }
}
```

- Suppose i needs to be spilled because of foo
- Since i and x have the same (unmodified) value, can simply reload x, rather than spilling
- Since switch is either 0 or 1, and i is in that range, there’s no need to have a branch bounds check

* Suppose there is a data race that causes another thread to update x to 5 while this thread is executing foo
To enable high-performance applications greater control of memory visibility, the C11 memory model provides support for low-level atomics, annotations on memory accesses, that allow for weaker visibility guarantees than provided by SC

- SC annotations enforce a total ordering on all read and write accesses
- Release (REL)/Acquire (ACQ) annotations enforce a causally-consistent ordering between the operations that precede the release and follow the acquire
- Relaxed (RLX) annotations compile to single hardware loads and stores with no additional synchronization other than basic coherence guarantees

Formalize these notions using relations defined over execution events that identify potential visibility properties based on the semantics of these annotations
Constraints on reads cannot be simply that they read from the “most recent” write since there is no global linear time in a concurrent execution. Use a happens-before relation (and relations derived from it) to reason about allowed behaviors.

In a sequential execution, the “sequenced-before” (sb) corresponds to happens-before.
Threads and Non-Atomic Accesses

```
void foo(int* p) {*p = 3;}
int main() {
    int x = 2;
    int y;
    thread t1(foo, &x);
    y = 3;
    t1.join();
    return 0; }
```

```
adjacent-synchronizes-with edges added between parent and child thread creation event
```
SC Atomics

- concurrent access to `x` no longer considered a data race
- SC atomic operations are totally ordered
  - form an interleaving in a global timeline
- Initialization of an atomic object by non-atomic stores can potentially race however
- All other accesses are through atomic stores and loads

```c
int main() {
    atomic_int x;
    x.store(2);
    int y = 0;
    {{ x.store(3);
      y = ((x.load())==3);
    }};
    return 0; }
```
Release-Acquire Synchronization

- **Message-passing program**
- **Desired behavior:**
  - Causal ordering between events that precede the release-write to y by the sender and the events that follow the acquire-read to y by the receiver

```plaintext
// sender
x = ...  // receiver
y = 1;
while (0 == y);
r = x;
```

(a). In the model, any instance of a read-acquire that reads from a location, (c), and (b) precedes (c) in the modification order (mo) sense) a write-acquire. The modification order of a candidate execution (here

![Diagram]

The desired guarantee here is that the receiver must see the data until the flag is set and then reads the data.

Pairs of a write-release and a read-acquire support the following

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

An atomic action must read a write that is in one of its

A write-release gives rise to a synchronization, from thread create etc. – 

The values that can be read by an atomic action depend on happens-before, in the sense below. More generally, the read-acquire can synthesize with a write-release (to the same location) that is before

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

The modification order and the sc order we saw earlier must also be consistent, in the following sense:

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

In the example, the release action (b) has a release sequence spanning multiple words) and then sets a flag

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.

∀ sc ∈ \[\emptyset, \text{mod} \ldots \text{rel}\] of actions but by a relation from the head to all the elements, as the reflexive edge from the head to itself.
Relaxed Atomics

- No additional synchronization edges introduced
- Only provides basic coherence guarantees

```c
int main() {
    int r1, r2;
    atomic_int x = 0;
    atomic_int y = 0;
    {{
        { r1 = x.load(mo_relaxed));
            y.store(r1,mo_relaxed);
        }
    |||
        { r2 = y.load(mo_relaxed));
            x.store(r2,mo_relaxed);
        }
    }}
    return 0; }
```

- The definition of relaxed atomics as strictly interpreted permits load buffering and out-of-thin-air reads
- The draft standard proposes conditions to prohibit such scenarios but at a cost of reasoning complexity
Coherence Rules

Forbidden executions
Examples

Message passing not sound with relaxed atomics:

![Diagram showing message passing with relaxed atomics]

Write-to-read causality guarantees not respected with relaxed atomics:

![Diagram showing write-to-read causality with relaxed atomics]

IRIW admitted:

![Diagram showing IRIW admitted]

Principles of Concurrency, Spring 2022
Compiler Transformations

Sequentialization

\[ C_1 \parallel C_2 \simrightarrow C_1 ; C_2 \]

\[
\begin{align*}
    r_1 &= x.\text{load}(\text{RLX}) ; \quad r_2 = y.\text{load}(\text{RLX}) ; \\
    y.\text{store}(1, \text{RLX}) ; &\quad x.\text{store}(1, \text{RLX}) ; \\
    r_1 &= r_2 = 1 \\
    a = 1 ; &\quad \begin{cases} 
      \text{if } (x.\text{load}(\text{RLX})) \\
      \text{if } (a) \\
      y.\text{store}(1, \text{RLX}) ; \\
      \text{if } (y.\text{load}(\text{RLX})) \\
      x.\text{store}(1, \text{RLX}) ; \\
    \end{cases} \\
\end{align*}
\]

- No consistent execution in which load of \( a \) occurs
- Now, apply sequentialization:

\[
\begin{align*}
    a = 1 ; &\quad \begin{cases} 
      \text{if } (x.\text{load}(\text{RLX})) \\
      \text{if } (a) \\
      y.\text{store}(1, \text{RLX}) ; \\
      \text{if } (y.\text{load}(\text{RLX})) \\
      x.\text{store}(1, \text{RLX}) ; \\
    \end{cases} \\
\end{align*}
\]

- An execution in which \( a = x = y = 1 \) is now permitted

Principles of Concurrency, Spring 2022
Compiler Transformations

- The only possible final state of the program is $a = z = 1$ and $x = y = 0$.
- Reordering the two stores in the first thread now allows an execution in which $a = z = x = y = 1$ is allowed.