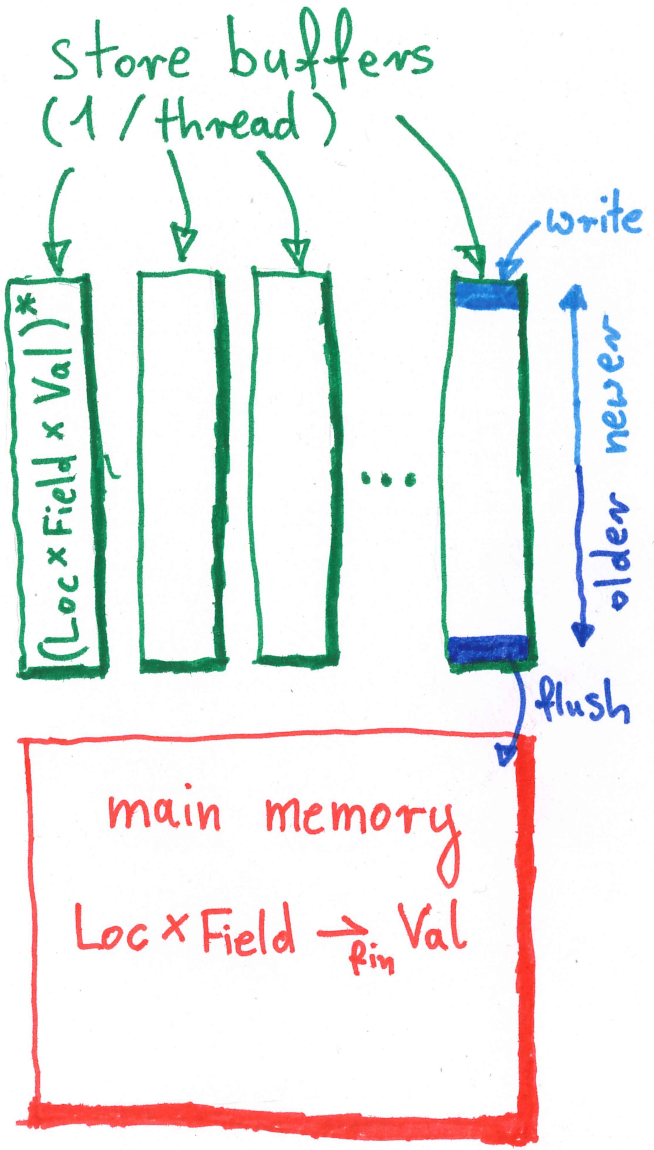
A Separation Logic for Fictional Sequential Consistency

Filip Sieczkowski, **Kasper Svendsen**, Lars Birkedal
Aarhus University

January 21, 2014

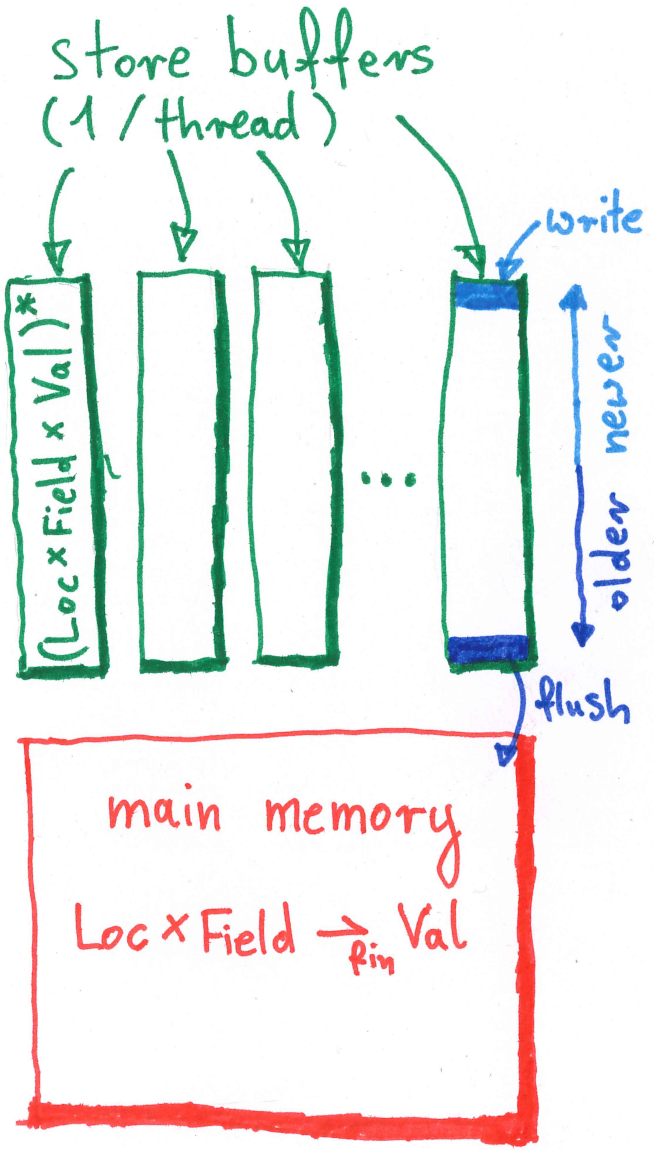
Weak Memory: x86-TSO

- Each processor is equipped with a FIFO buffer.
- Writes are queued in the buffer.
- Threads first read from own buffer before consulting main memory.
- Buffered writes are flushed to main memory non-deterministicly.
- CAS'ing flushes buffer.

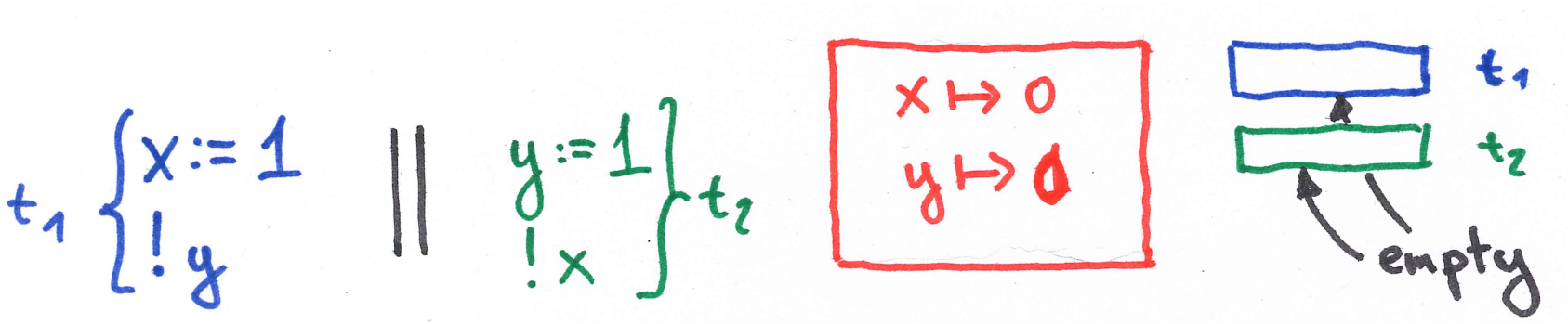


Weak Memory: x86-TSO

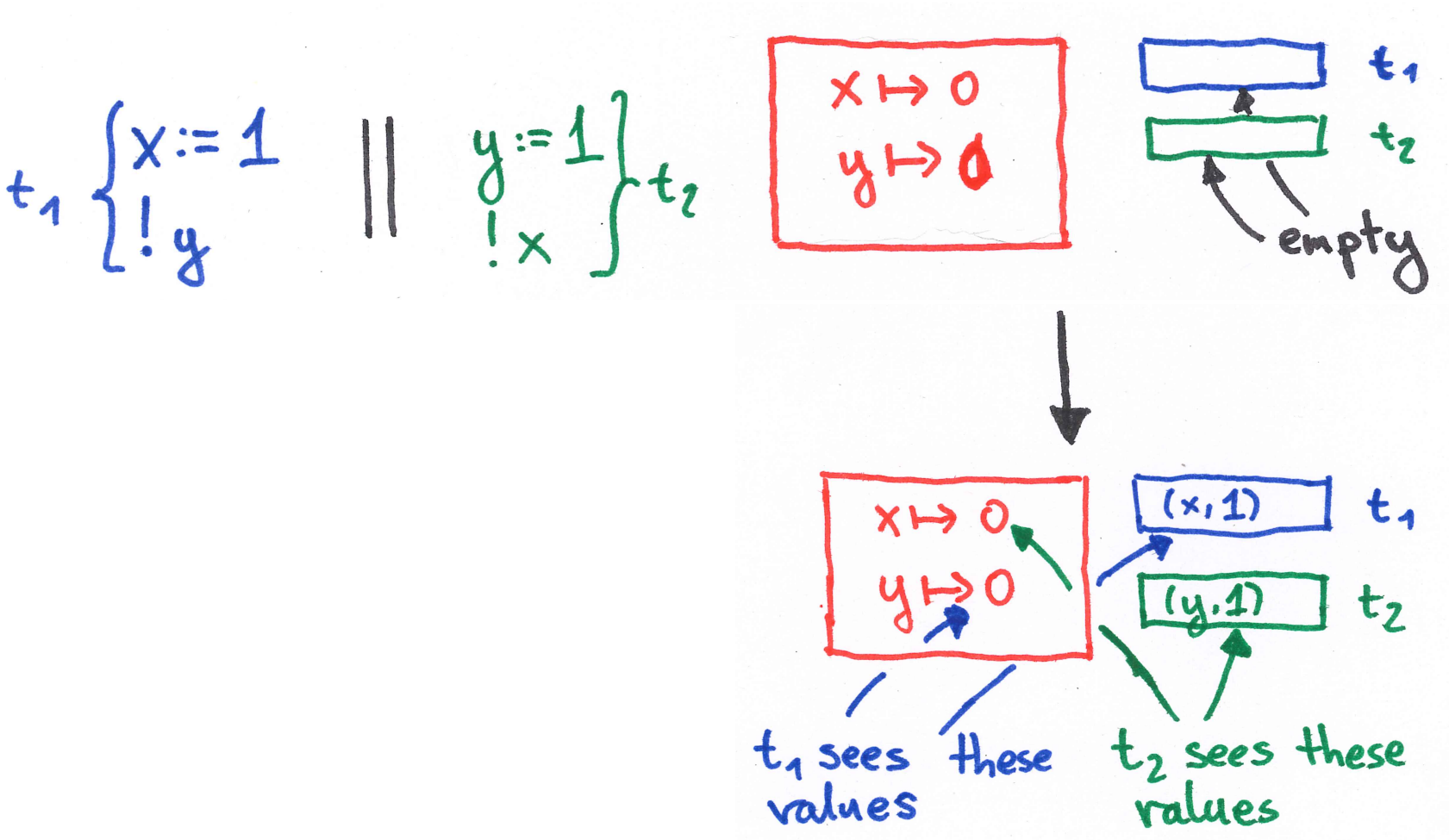
- Each thread is equipped with a FIFO buffer.
- Writes are queued in the buffer.
- Threads first read from own buffer before consulting main memory.
- Buffered writes are flushed to main memory non-deterministicly.
- CAS'ing flushes buffer.



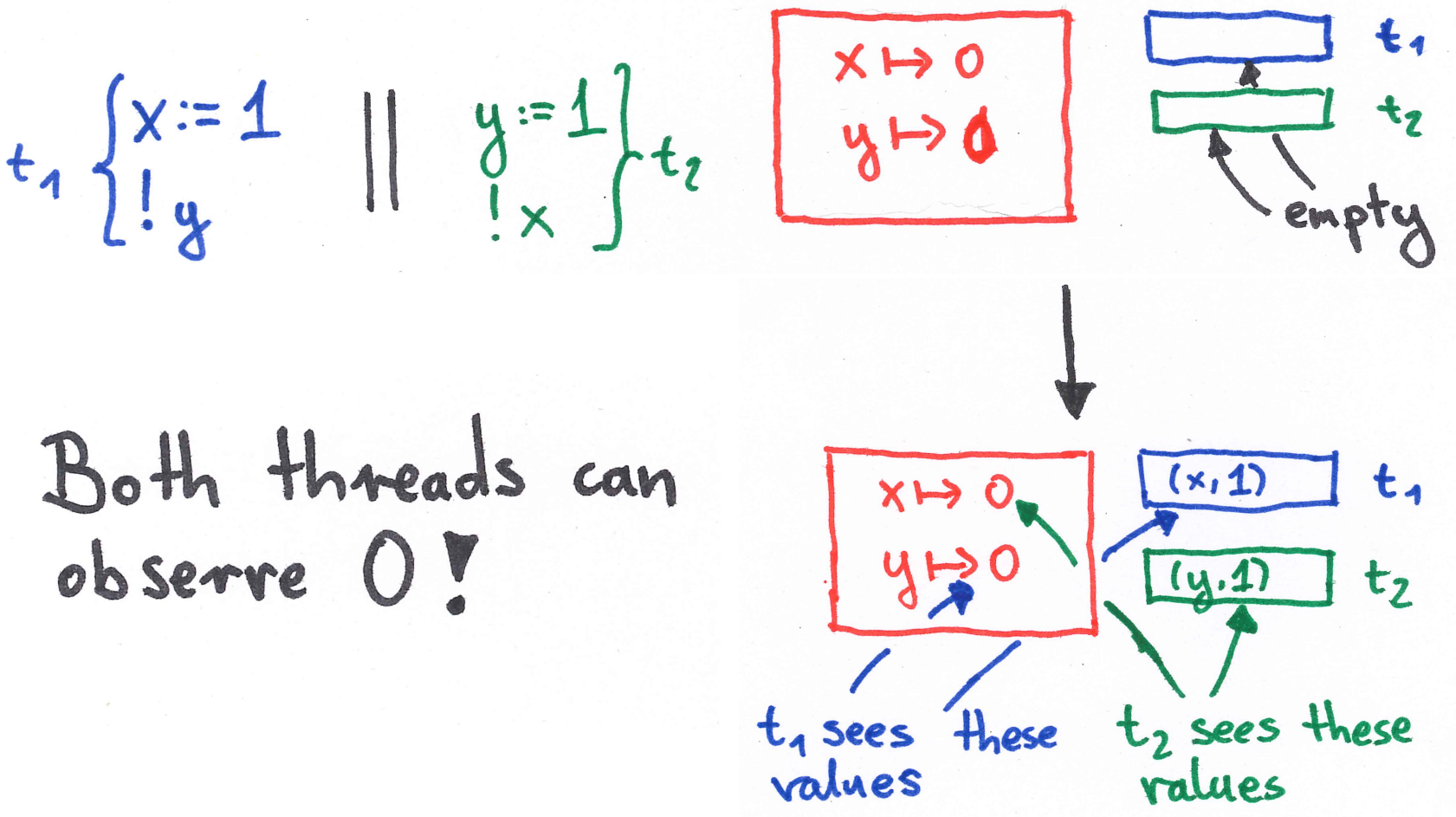
Weak behaviors on x86-TSO



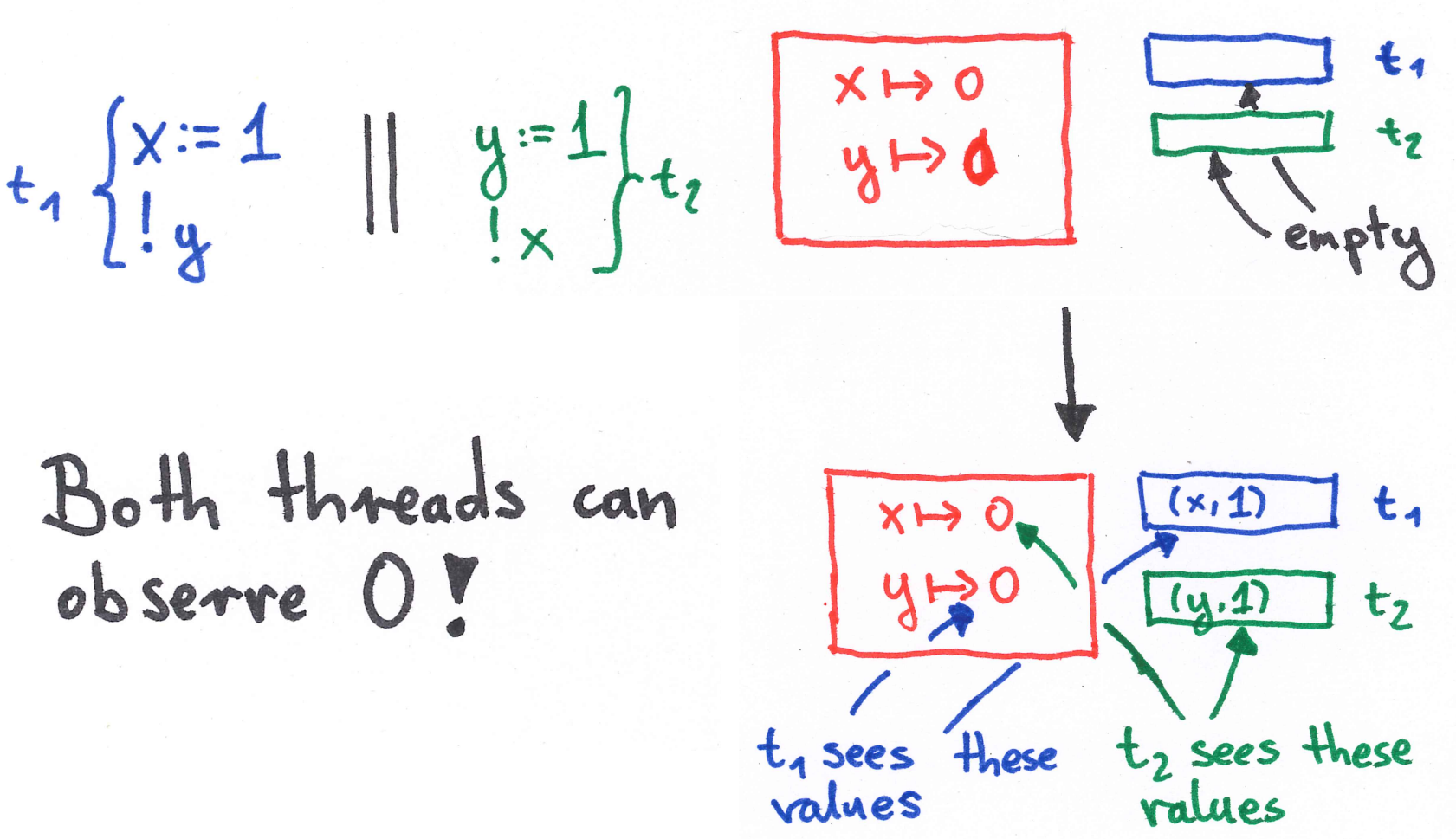
Weak behaviors on x86-TSO



Weak behaviors on x86-TSO



Weak behaviors on x86-TSO



- Each thread has a **subjective** view of the state.

Reasoning about TSO

- Can extend SL to TSO setting by internalizing buffers
- However, for **most** code we should not have to reason about buffers!
  - E.g., any TSO execution of a spin-lock well-synchronized x86-program can be simulated by an SC machine.

    - Owens, ECOOP'10.

Reasoning about TSO

Goal

- A proof system that allows for explicitly reasoning about buffers, when we have to, and allows for standard separation logic reasoning when we do not.

- Reasoning about fine-grained concurrent data structures and synchronization primitives will require low-level reasoning about buffers, but reasoning about **most** clients should not.

Reasoning about TSO

Our solution

- A proof system with **two** interconnected separation logics
  - The TSO logic for low-level reasoning about buffers
  - The SC logic for high-level reasoning about clients
- **Fiction of sequential consistency:** Racy code with weak behaviours may still provide an SC specification

The SC logic

- In the SC logic we interpret the pre- and postcondition from the perspective of the thread we are reasoning about
- $x.f \mapsto v$ holds from the perspective of thread t if
  - the most recent update to $x.f$ in t's buffer is v and no other threads have buffered updates to $x.f$
  - or, $x.f$ is v in main memory and there are no buffered updates to $x.f$ any store buffer

The SC logic

- Usual separation logic assertions and proof rules

$$\frac{}{[x.f \mapsto v]\ x.f\ [\lambda r.\ x.f \mapsto v * r = v]}\ \text{S-Read}$$

$$\frac{}{[x.f \mapsto \_]\ x.f := v\ [\lambda \_.\ x.f \mapsto v]}\ \text{S-Write}$$

The SC logic

- Usual separation logic assertions and proof rules

$$\frac{}{[x.f \mapsto v]\ x.f\ [\lambda r.\ x.f \mapsto v * r = v]}\ \text{S-Read}$$

$$\frac{}{[x.f \mapsto \_]\ x.f := v\ [\lambda \_.\ x.f \mapsto v]}\ \text{S-Write}$$

- However, to transfer a resource between two threads, their perspective of the resource must match.

The SC logic: Transferring resources

- This SC lock specification allows us transfer resources using the resource invariant R:

$$\begin{array}{rll} [R] & \texttt{new Lock()} & [\lambda r.\ \mathit{isLock}(r, R)] \\ [\mathit{isLock}(\mathit{this}, R)] & \texttt{Lock.Acq()} & [\lambda \_.\ \mathit{locked}(\mathit{this}, R) * R] \\ [\mathit{locked}(\mathit{this}, R) * R] & \texttt{Lock.Rel()} & [\lambda \_.\ \mathit{emp}] \end{array}$$

$$\mathit{isLock}(x, R) \Leftrightarrow \mathit{isLock}(x, R) * \mathit{isLock}(x, R)$$

- We release and acquire ownership of R from the perspective of the acquiring/releasing thread.

The SC logic: Transferring resources

- This SC lock specification allows us transfer resources using the resource invariant R:

$$
\begin{array}{rll}
[R] & \texttt{new Lock()} & [\lambda r.\ \mathit{isLock}(r, R)] \\
[\mathit{isLock}(\mathit{this}, R)] & \texttt{Lock.Acq()} & [\lambda \_.\ \mathit{locked}(\mathit{this}, R) * R] \\
[\mathit{locked}(\mathit{this}, R) * R] & \texttt{Lock.Rel()} & [\lambda \_.\ \mathit{emp}]
\end{array}
$$

$$
\mathit{isLock}(x, R) \Leftrightarrow \mathit{isLock}(x, R) * \mathit{isLock}(x, R)
$$

- We release and acquire ownership of R from the perspective of the acquiring/releasing thread.

The SC logic: Transferring resources

- To verify a lock implementation we have to prove the implementation ensures the perspective of the releasing and acquiring thread match.
  - This requires low-level reasoning in the TSO logic.
- Once we have verified such a lock, the SC logic (roughly) generalizes Concurrent SL to a TSO setting.

Transferring resources: A spin-lock

- This lock implementation ensures the perspectives match.

```
class Lock {
  bool locked := false;

  Acq() {
    let x = CAS(this.locked, true, false) in
      if x then () else Acq()
  }

  Rel() {
    this.locked := false
  }
}
```

Transferring resources: A spin-lock

- This lock implementation ensures the perspectives match.

```
class Lock {
  bool locked := false;

  Acq() {
    let x = CAS(this.locked, true, false) in
      if x then () else Acq()
  }

  Rel() {
    this.locked := false
  }
}
```

Once this buffered release makes it to main memory, any prior buffered writes have already been flushed.

The TSO logic

- A different set of Hoare triples

$$\{\lambda t.\ P(t)\}\ e\ \{\lambda t.\ \lambda r.\ Q(t)(r)\}$$

  and non-standard assertions for reasoning about buffers

- Basic TSO assertions constructed through embeddings:
  - $\lceil P \rceil \sim P$ holds in main memory and there are no buffered updates to the state asserted by P
  - $\lceil P\ \mathit{in}\ t \rceil \sim P$ holds from the perspective of thread t

The TSO logic

Relating the SC and TSO logic

- Embedding allows us to move between logics:

$$\frac{[P]\ e\ [\lambda r.\ Q(r)]}{\{\lambda t.\ \lceil P\ in\ t\rceil\}\ e\ \{\lambda t.\ \lambda r.\ \lceil Q(r)\ in\ t\rceil\}}$$

- and explain when we can transfer resources:

$$\forall t.\ \lceil R\rceil \Rightarrow \lceil R\ in\ t\rceil$$

The TSO logic

- An "until" operator to express ordering dependencies:

$$P\ \mathcal{U}_t\ Q \quad \sim$$

there exists an buffered update in t's buffer such that P holds before the update and Q holds once this buffered update has been flushed

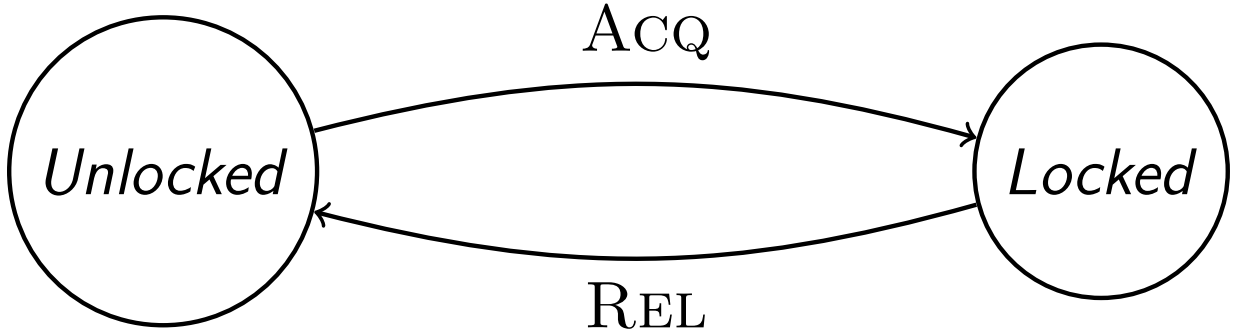
- The "until" operator describes buffered writes:

$$\{\lambda t.\ \lceil x.f \mapsto \mathit{true} \rceil * \lceil R\ \mathit{in}\ t \rceil\}$$
$$x.f := \mathit{false}$$
$$\{\lambda t.\ \lambda\_.\ \lceil x.f \mapsto \mathit{true} \rceil\ \mathcal{U}_t\ \lceil x.f \mapsto \mathit{false} * R \rceil\}$$

Conclusion

- A higher-order separation logic for a language with a TSO memory model that supports:
  - low-level reasoning about synchronization primitives and fine-grained concurrent data structures
  - a **fiction of sequential consistency** that allows us to give SC specifications to certain racy implementations

Verifying the spin-lock



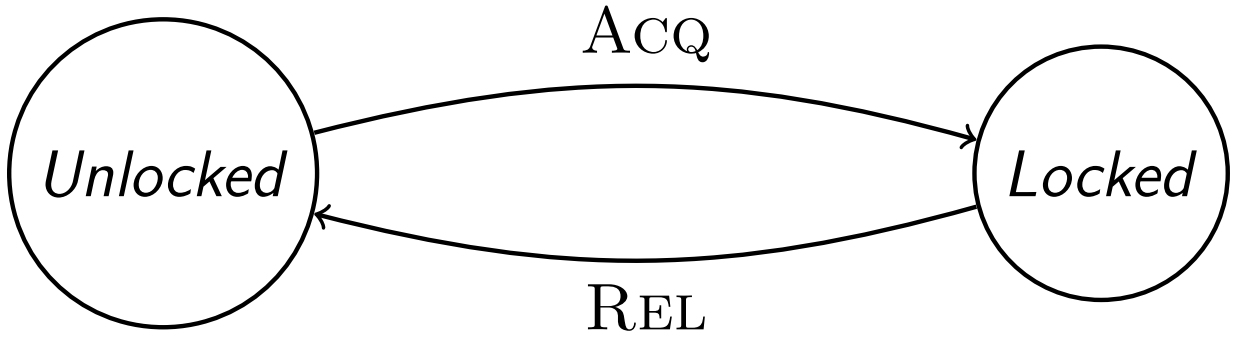
$$I_{Locked}(x, R, \mathsf{n}) = \lceil x.\mathsf{locked} \mapsto \mathsf{true} \rceil$$

$$I_{Unlocked}(x, R, \mathsf{n}) =$$

$$\lceil x.\mathsf{locked} \mapsto \mathsf{false} * R * ... \rceil \vee$$

$$\exists \mathsf{t}.\ \lceil x.\mathsf{locked} \mapsto \mathsf{true} \rceil\ \mathcal{U}_t\ \lceil x.\mathsf{locked} \mapsto \mathsf{false} * R * ... \rceil$$

Verifying the spin-lock



$$I_{Locked}(x, R, \mathsf{n}) = \lceil x.\mathsf{locked} \mapsto \mathsf{true} \rceil$$

$$I_{Unlocked}(x, R, \mathsf{n}) =$$

$$\lceil x.\mathsf{locked} \mapsto \mathsf{false} * R * \ldots \rceil \vee$$

$$\exists \mathsf{t}.\ \lceil x.\mathsf{locked} \mapsto \mathsf{true} \rceil\ \mathcal{U}_t\ \lceil x.\mathsf{locked} \mapsto \mathsf{false} * R * \ldots \rceil$$