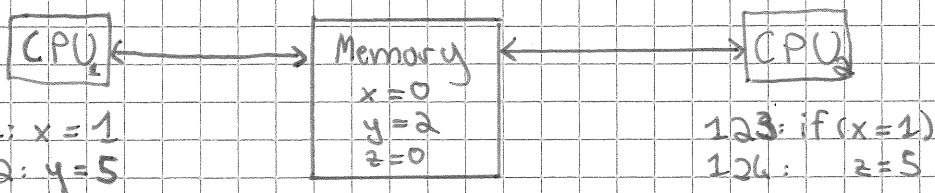
Total Store Ordering:



In a perfect world...

- → CPU writes x=1 and directly stores this in the Memory / RAM.
- → every other CPU sees x=1 immediately.
- → Sequential consistency - Strong Memory Model

The truth:

- → memory accesses are slow: clock-rate(CPU) ≥ 10 · clock-rate(RAM).
- → do not immediately write to memory!
  ↳ if we would, parallel programming very slow :(
- → allow more behavior - weak Memory Model.

On x86 architecture:

- → Writes of CPU are put into a buffer.
- → if there are some writes, flush to memory
- → other CPUs may see changes to variables only later...

Parallel Programs:

Definition: Parallel programs with shared memory are defined by the following grammar:

<prog> ::= program <name> <thread>* // name and finite list of threads

<thread> ::= thread <threadid> // identifier
regs <reg>* // list of local registers
init <label> // label of initial instruction
begin <inst>* end // list of labeled instructions

<inst> ::= <label>: <inst> | goto <label>

```
<inst> ::= <reg> ← mem[<reg>]      // Load from memory into local register
         | mem[<reg>] ← <reg>      // store
         | mfence                  // memory fence
         | <reg> ← <expr>          // local assignment
         | assert <expr>           // assertion

<expr> ::= fun(<reg>*)             // function on fin. many registers.
```

Programs have a finite <u>data domain</u> DOM, and a fin. <u>function domain</u> FUN.

We assume:

- The thread identifiers <threadid> are distinct numbers.
- The registers <reg> are from a finite set of names (x, y, z, ...) and they are not shared among threads.
- The labels <label> are strings and each command <inst> has a distinct label.
- The elements of DOM can be used as register content and memory addresses.
- $0 \in DOM$.
- FUN are functions over DOM, with multiple input parameters. Functions are computable.
- Each thread only jumps to its own labels and only accesses its own registers.

TSO Semantics:

<u>Example:</u> Dekker's mutex

l_0: mem[x] ← 1; goto l_1;	l_0': mem[y] ← 1; goto l_1';
l_1: r ← mem[y]; goto l_2;	l_1': r' ← mem[x]; goto l_2';
l_2: assert r == 0; goto l_3;	l_2': assert r' == 0; goto l_3';
l_3: ... // critical section	l_3': ... // critical section

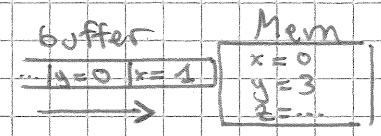
→ Simplified notation. Can easily be transformed to above grammar.

→ Under sequential consistency: at most one thread can enter critical section.

③

Idea of TSO:

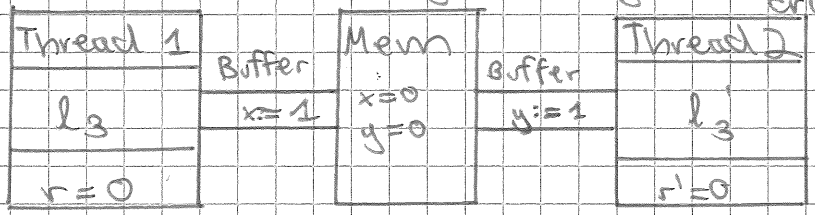
- Each thread has a store buffer. Stores are buffered locally and later flushed to the memory in FIFO manner.
- Stores in buffer are not visible to other threads.
- A thread can read stores from its buffer: early read.



Example: Reconsider Dekker's mutex.

This time, we run it under TSO.

⇒ Get unwanted behavior:
- Both threads issue the store and put it into their buffer.
- Both threads load value 0 in r, r'. Taken from memory.
- Both asserts go through ⇒ both threads in critical section!



TSO Semantics:

We define configurations and the TSO-transition relation.

Definition: TSO configurations

Fix program P with threads $t_1, \dots, t_n$.
Assume thread t_i has
- identifier i,
- initial label $l_{0,i}$,
- registers R_i.

Use $TID := \{1, \dots, n\}$ for all thread identifiers,

$LAB :=$ all labels used by all threads,

$VAR := DOM \cup \bigcup_{i=1}^{n} R_i$ for all locations (addresses and registers).

A <u>configuration</u> of P is an element from

$$CF = LAB^{TID} \times DOM^{VAR} \times (DOM \times DOM)^{*TID},$$

it is a tuple $cf = (pc, val, buf)$, where

- $pc : TID \to LAB$ is the <u>program counter</u>,
- $val : VAR \to DOM$ is the <u>valuation</u>, and
- $buf : TID \to (DOM \times DOM)^*$ a collection of <u>local buffers</u>.

$buf(i) = \underbrace{(a_1, v_1) \dots (a_k, v_k)}$

means: value v_1 is stored in address a_1.

The <u>initial configuration</u> is $cf_0 = (pc_0, val_0, buf_0)$ with

$pc_0(i) = l_{0,i} \quad \forall i \in TID,$

$val_0(x) = 0 \quad \forall x \in VAR,$

$buf_0(i) = \varepsilon \quad \forall i \in TID.$

<u>Definition:</u> TSO transition relation.

Assume, we are in configuration $cf = (pc, val, buf)$ and $pc(i) = l$ for some thread i.
We want to execute $l : \langle inst \rangle ; \text{goto } l'$.

Define $pc' := pc[i := l']$

The TSO transition relation $\to_{TSO} \subseteq CF \times CF$ is the smallest relation that satisfies the following rules:

$$\text{(FEAR(Y))} \quad \frac{\langle inst \rangle = r \leftarrow mem[r'],\ a = val(r'),\ buf(i)\!\downarrow_a = (a := v).\beta}{(pc, val, buf) \longrightarrow_{TSO} (pc', val[r := v], buf)}$$

- $buf(i)\!\downarrow_a$ is the projection of $buf(i)$ to stores to address a,
- $(a := v)$ means that v is stored to address a.

$$\text{(LOAD)} \quad \frac{\langle inst \rangle = r \leftarrow mem[r'],\ a = val(r'),\ buf(i)\!\downarrow_a = \varepsilon,\ v = val(a)}{(pc, val, buf) \longrightarrow_{TSO} (pc', val[r := v], buf)}$$

- $buf(i)$ contains no store on address a → read from memory.

$$\text{(STORE)} \quad \frac{\langle inst \rangle = mem[r] \leftarrow r',\ a = val(r),\ v = val(r')}{(pc, val, buf) \longrightarrow_{TSO} (pc', val, buf[i := (a := v).buf(i)])}$$

- issue a store → append to buffer

⑤ (UPDATE) $$\frac{buf(i) = \beta.(a=v)}{(pc, val, buf) \longrightarrow_{TSO} (pc, val[a:=v], buf[i:=\beta])}$$

- Move earliest write in buffer to memory

(MFENCE) $$\frac{\langle inst\rangle = mfence,\ buf(i) = \varepsilon}{(pc, val, buf) \longrightarrow_{TSO} (pc', val, buf)}$$

- Blocks thread until buffer is flushed to memory completely.

(ASSERT) $$\frac{\langle inst\rangle = assert\ e,\ [\![e]\!] \neq 0}{(pc, val, buf) \longrightarrow_{TSO} (pc', val, buf)}$$

- assert only continues execution if $[\![e]\!] \neq 0$.
- $[\![e]\!]$ is the evaluated expression, where registers r are replaced by val(r).

~~(FENCE)~~

(ASSIGN) $$\frac{\langle inst\rangle = r \leftarrow e,\ [\![e]\!] = v}{(pc, val, buf) \longrightarrow_{TSO} (pc', val[r:=v], buf)}$$

- Local assignment changes register content.

TSO Reachability:

We want to solve the following problem:

TSO Reachability
Given: Program P, program counter p
Quest: Is there a computation $cf_0 \longrightarrow^*_{TSO} (pc, val, buf)$ for some val and buf?

Goal: Show that problem is decidable

Idea: Construct LCS L_p simulating P and use that reachability on LCSs is decidable.

Approach: Construct LCS $L_p^0, L_p^1, L_p^2, L_p^3$, and L_p.
⋮
Each L_p^i fixes problems from L_p^{i-1}.

STEP 1: From P to L_p^0:

→ Shared memory communication is like lossiness: A store might get overwritten before it is seen by another thread.

→ Understand TSO buffers as lossy channels.

$\Rightarrow$ Construct L_p^0 with: states $LAB^{TID} \times DOM^{VAR}$

a channels for each thread over $DOM \times DOM$

transition relation induced by $\rightarrow_{TSO}$.

STEP 1: Towards L_p^1:

Problem: Well quasi ordering for LCS is $\subseteq^*$ Higman's ordering.

$\rightsquigarrow$ Not a simulation relation for TSO.

l_0: r ← mem[x]; || l_0': mem[y] ← 1;
l_1: assert (r=1); || l_1': mem[x] ← 1;
l_2: r ← mem[y]; || l_2'; ...
l_3: assert (r=0); ||
l_4: ...;

Configuration of L_p^0 (also a TSO config.)

- $cf = (\underbrace{l_0, l_2'}_{pc}, \underbrace{\overset{x=0}{y=0}}_{val}, \underbrace{(\varepsilon, x=1 . y=1)}_{buf})$
- $cf' = (l_0, l_2', \overset{x=0}{y=0}, (\varepsilon, x=1))$.

Then $cf \xrightarrow{\nu_1} cf'$. Conf. cf' can reach $(l_4, l_2', \overset{x=1}{y=0}, \varepsilon) = \gamma$

But cf cannot reach a conf. with $(l_4, l_2', \overset{x=1}{y=0}, w)$

$\xrightarrow{\nu_1}$

$(l_4, l_2', \overset{x=1}{y=0}, \varepsilon)$.

$\Rightarrow$ No sim. relation: $cf \not\rightarrow \gamma'$. $cf \xrightarrow{\nu_1} \gamma'$. $cf' \longrightarrow \gamma$

$\Rightarrow$ Lossiness gives inconsistent memory configurations. $\binom{x=1}{y=0}$.

Fix: In L_p^1, [redacted] can issueing sends a memory snapshot via the channel. Contains values for all memory addresses.

Problem vanishes: $(\overbrace{l_0, l_2'}^{pc}, \overbrace{\binom{x=0}{y=0}}^{val}, (\varepsilon, \binom{x=1}{y=1}, \binom{x=0}{y=1}))$

and $(l_0, l_2', \binom{x=0}{y=0}, (\varepsilon, \binom{x=1}{y=0}))$

corresponding to cf and cf' are incomparable now.

STEP 2: Towards L_p^2:

Problem: L_p^1 allows behavior not possible under TSO.

l_0: mem[y] ← 0; || l_0': mem[x] ← 1;
l_1: ... || l_1': r ← mem[x];
|| l_2': assert (r == 0);
|| l_3': ...

→ (l_1, l_3') is not reachable under TSO;
We can only load 1 from x, since store in l_0' has been performed.

→ But a configuration with $(l_1, l_3', \_\_)$ is reachable in L_p^1:

$(l_0, l_0', \begin{pmatrix} x=0 \\ y=0 \end{pmatrix}, (\varepsilon, \varepsilon))$
$\rightarrow_{TSO}^{2} (\{l_1, l_1'\}, \begin{pmatrix} x=0 \\ y=0 \end{pmatrix}, (\begin{pmatrix} x=0 \\ y=0 \end{pmatrix}, \begin{pmatrix} x=1 \\ y=0 \end{pmatrix}))$ // buffer stores
$\rightarrow_{TSO} (\{l_1, l_1'\}, \begin{pmatrix} x=1 \\ y=0 \end{pmatrix}, (\begin{pmatrix} x=0 \\ y=0 \end{pmatrix}, \varepsilon))$ // update
$\rightarrow_{TSO} (l_1, l_1', \begin{pmatrix} x=0 \\ y=0 \end{pmatrix}, (\varepsilon, \varepsilon))$ // update
$\rightarrow_{TSO}^{2} (l_1, l_3', \begin{pmatrix} x=0 \\ y=0 \end{pmatrix}, (\varepsilon, \varepsilon))$.

⇒ Threads do not synchronize on memory updates.
Use values that are no longer in mem.

Fix: All threads share the same buffer.

In our example we would get:

$(l_1, l_1', \begin{pmatrix} x=0 \\ y=0 \end{pmatrix}, \underbrace{\begin{pmatrix} x=1 \\ y=0 \end{pmatrix}}_{\text{store of y takes x=1 into account.}}, \begin{pmatrix} x=1 \\ y=0 \end{pmatrix})$

STEP 3: Towards L_p^3:

Problem: L_p^2 allows less behavior than TSO.

mem[x] ← 1; ① ② || r_2 ← mem[y]; ⑫ || mem[y] ← 1; ③ ⑭ || r_4 ← mem[x]; ⑧
mem[x] ← 2; ⑥ ⑦ || assert r_2 == 2; ⑬ || r_3 ← mem[x]; ④ || assert r_4 == 2; ⑨
⋮ || r_2 ← mem[y]; ⑮ || assert r_3 == 1; ⑤ || mem[y] ← 2; ⑩ ㉑
|| assert r_2 == 1; ⑯

This behavior is not possible in L_p^2:

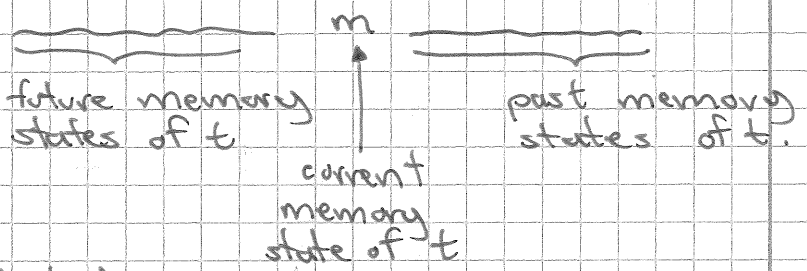
- All threads use one channel / buffer.
- So, $y=1$ will be sent to memory before $y=2$.

→ In L_p^2, memory updates in the buffer are delivered to the memory in the order in which they ~~occur~~ entered the buffer.

⇒ In t_2, it is not possible to load $y=2$ before $y=1$.

→ In TSO, memory updates can have any order if they stem from different threads.

Fix: Add for each thread, a pointer to a position inside the buffer.

For thread t: buf = 

Then, updates are simulated by moving the pointer to the left. Looks like:

$$\begin{pmatrix} x=2 \\ y=1 \end{pmatrix}\cdot\begin{pmatrix} x=2 \\ y=2 \end{pmatrix}\cdot\begin{pmatrix} x=2 \\ y=0 \end{pmatrix}\cdot\begin{pmatrix} x=1 \\ y=0 \end{pmatrix}\cdot\begin{pmatrix} x=0 \\ y=0 \end{pmatrix}$$

(t_2, t_4, t_3, t_1)

STEP 4: Towards L_p:

Problem: In L_p^3, early reads are not modeled.

Fix: Remember the last write of a thread to an address.

$$\begin{pmatrix} x=2 \\ y=1 \end{pmatrix},(t_3,y)\cdot\begin{pmatrix} x=2 \\ y=2 \end{pmatrix},(t_4,y)\cdot\begin{pmatrix} x=2 \\ y=0 \end{pmatrix},(t_1,x)\cdot\begin{pmatrix} x=1 \\ y=0 \end{pmatrix},(t_3,x)\cdot\begin{pmatrix} x=0 \\ y=0 \end{pmatrix},-$$

(t_2, t_4, t_3, t_1)

Formal Construction: Let P be a program.

The LCS with strong symbols L_p is defined by

$$L_p = (Q, q_0, C, M, \Sigma, \to),$$

Σ: cannot be lost but are bounded.

where • $Q = \underbrace{LAB^{TID}}_{pc} \times \underbrace{DOM^{Var}}_{val}$

• $C = \{buf\}$,

• $M = DOM \xrightarrow{DOM} NOT\ DOM^{VAR}$, as local registers are not in the shared memory.

• $S = \Big(DOM^{DOM} \times \underbrace{(TID \times DOM \cup \{\varepsilon\})}_{\text{last write}} \times \underbrace{\mathcal{P}(TID)}_{\text{threads pointing to snapshot } t} \Big) \setminus \underbrace{DOM^{DOM} \times \{\varepsilon\} \times \{\emptyset\}}_{\text{normal mem. states/updates. can get lost.}}$

Remarks:

The number of strong symbols that can occur is bounded by $|TID| \cdot |DOM| \cdot |TID|$.

→ For each $d \in DOM$, there are at most $|TID|$ last writes to it, and for each thread, we need one pointer.

Definition: Transition relation of L_p:

In a state $(pc, \underbrace{val}_{\substack{\text{valuation of local registers and copy of last mem. valuation sent to buffer.}}}) \in Q$, with $pc(t) = l$.

} memory content when complete buffer was flushed

We describe transitions that are realized by using sequences of transitions.

Store: If $l: mem[r] \leftarrow r'$; goto l':

Let $val(r) = d$, $val(r') = v$.

$(pc, val) \xrightarrow{buf \{(mval, \varepsilon, tidset) / (mval, (t,a), tidset)\}} *$

realizable with CS transitions:

means: check if buf contains strong symbol $(mval, (t,a), tidset)$ with last write (t,a).
→ Replace it by $(mval, \varepsilon, tidset)$. Last write to a of t will be new after transition.

$* \xrightarrow{buf!(\underbrace{val_{DOM}[a:=v]}_{\substack{\text{memory snapshot with updated } a}}, (a,t), \emptyset)} (pc[t=l'], val[a=v])$.

Load: If $l: r \leftarrow mem[r']; goto\ l';$ and $val(r') = a$.

Two cases:

Early read:

$$(pc, val) \xrightarrow{assert\ (mval, (t,a), tidset) \in buf} (pc[t = l'], val[r = mval(a)]).$$

Load from memory:

$$(pc, val) \xrightarrow[\text{and } (mval, *, \{t\} \cup tidset) \in buf]{assert\ (mval', (t,a), tidset') \notin buf} (pc[t = l'], val[r = mval(a)]).$$

Fence: If $l: mfence; goto\ l';$

$$(pc, val) \xrightarrow{assert\ (mval, *, \{t\} \cup tidset) = head(buf)} (pc[t = l'], val).$$

Update:

$$(pc, val) \xrightarrow[\text{modify } buf = w_1 \cdot \hat{m}_1^{\$} \cdot \hat{m}_2 \cdot w_2]{assert\ buf = w_1 \cdot m_1 \cdot m_2 \cdot w_2} (pc, val),$$

with: $m_1 = (mval_1, last_1, tidset_1)$,

$m_2 = (mval_2, last_2, \{t\} \cup tidset_2)$ // current mem. state of t

$m_1' = (mval_1, last_1 \setminus \{(t, *)\}, tidset_1 \cup \{t\})$

$m_2' = (mval_2, last_2, tidset_2)$.

Assert: If $l: assert\ e; goto\ l';$

$$(pc, val) \xrightarrow{assert\ [\![e]\!] \neq 0} (pc[t = l'], val)$$

- does not involve buffer
- $[\![e]\!]$ is computed from val

Assign: If $l: r \leftarrow e; goto\ l';$

$$(pc, val) \rightarrow (pc[t = l'], val[r = [\![e]\!]]).$$

Theorem: Atig, Bouajjani, Burckhardt, Musuvathi 2010/12/16

Consider a program P. One can construct an LCS (with strong symbols) L_P such that a control state

pc is reachable in P, executed under TSO $\Leftrightarrow$ pc is reachable in L_P.

Hence, control state reachability for TSO is decidable.

Why is this correct? We give an idea.

Let A_1, A_2 be NFAs over disjoint alphabets Σ_1, Σ_2: $\Sigma_1 \cap \Sigma_2 = \emptyset$.
The shuffle is the language of all interleavings:

$$L(A_1) \sqcup L(A_2) = \{ w \in (\Sigma_1 \cup \Sigma_2)^* \mid w|_{\Sigma_1} \in L(A_1) \wedge w|_{\Sigma_2} \in L(A_2) \}.$$

We construct an automaton for the shuffle:

- Add to A_1 the following transitions
  For each $q \in Q_{A_1}$, $a \in \Sigma_2$ add $q \xrightarrow{a} q$.
- Similarly for A_2, where we add loops over Σ_1.

⇒ We obtain A_1' and A_2' over $\Sigma_1 \cup \Sigma_2$ and we have:

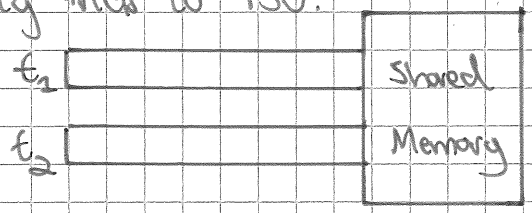
$$L(A_1') \cap L(A_2') = L(A_1) \sqcup L(A_2).$$

So the product $A_1' \times A_2'$ accepts the shuffle.

Example: $\{ab\} \sqcup \{xyz\}$

$$= L\left(\circlearrowright_{x,y,z} \xrightarrow{a} \circ \xrightarrow{b} \odot \right) \cap L\left(\circlearrowright_{a,b} \xrightarrow{x} \circlearrowright_{a,b} \xrightarrow{y} \circlearrowright_{a,b} \xrightarrow{z} \odot \circlearrowright_{a,b} \right)$$

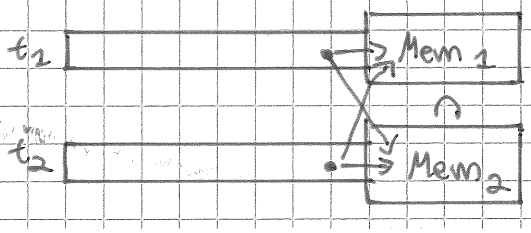
Apply tricks to TSO:



Memory sees a shuffle of the stores of both threads.

The stores by t_i arrive in order, but are interleaved with stores from other threads.

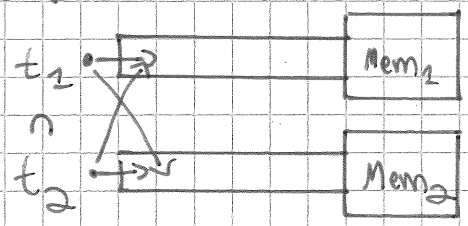
Use above tricks: assume each thread has its own memory. Add loops to each that produces arbitrary writes of other threads. Then intersect



The channels / buffers are FIFO.

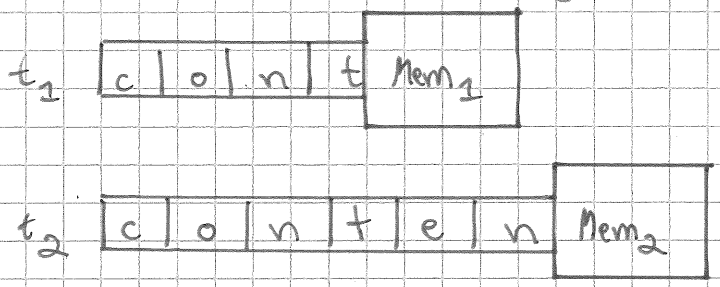
⇒ The stores leave the buffer in the order in which they were put in.

Instead of guessing the stores of the other thread at the memory, we let each thread guess these stores when inserting commands in the buffer:



Now the content of both buffers is the same.

But Mem_1 and Mem_2 may process at different speed:



This can be understood as one buffer with two pointers