Distributed Operating Systems
Memory Consistency & Cache Coherence

Till Smejkal

(slides by Julian Stecklina, Marcus Völp)
Symmetric Multiprocessor (SMP)

CPU0

CPU1

CPU2

CPU3

Memory

Processors

Bus or Crossbar

Shared Memory
Symmetric Multiprocessor (SMP)

- Processors
- Local Caches
- Bus or Crossbar
- Shared Memory

Distributed Operating Systems
Chip Multiprocessor (CMP)

- Processors
- Local Cache
- Shared LL-Cache
- Bus or Crossbar
- Shared Memory

Diagram showing a chip multiprocessor with four processors (CPU0, CPU1, CPU2, CPU3) each with L1 and L2 caches, connected to a central Memory via a bus or crossbar.
Cache Coherency

- Caches lead to multiple copies for the content of a single memory location

- Cache Coherency keeps copies “consistent”
  - locate all copies
  - invalidate/update content

- **Write Propagation**
  writes must eventually become visible to all processors.

- **Write Serialization**
  every processor should see writes to the same location in the same order.
Single-Writer, Multiple-Reader Invariant
For any memory location A, at any given time, either only a single core may write (or read-modify-write) the content of A or any number of cores may read the content of A.

Data-Value Invariant
The value of a memory location at the start of an operation is the same as the value at the end of its last write (read-modify-write) operation.

[based on Sorin et al., 2011]
Cache Lines

Distributed Operating Systems
Attempt 1: write through all caches
Attempt 1: write through all caches

CPU0: read x
x=0 stored in cache
Attempt 1: write through all caches

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache
Attempt 1: write through all caches

CPU0: read $x$
   $x=0$ stored in cache

CPU1: read $x$
   $x=0$ stored in cache

CPU0: write $x=1$
   $x=1$ stored in cache
   $x=1$ stored in memory
Attempt 1: write through all caches

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache

CPU0: write x=1
x=1 stored in cache
x=1 stored in memory

CPU1: read x
x=0 retrieved from cache
Attempt 1: write through all caches

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache

CPU0: write x=1
x=1 stored in cache
x=1 stored in memory

CPU1: read x
x=0 retrieved from cache

Write not visible to CPU1!
Attempt 2: write back

- CPU0
  - WB Cache
  - Memory
  - x=0

- CPU1
  - WB Cache
  - Memory
  - x=0
Attempt 2: write back

CPU0: read x
x=0 stored in cache
**Attempt 2: write back**

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache
Attempt 2: write back

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache

CPU0: write x=1
x=1 stored in cache
Attempt 2: write back

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache

CPU0: write x=1
x=1 stored in cache

CPU1: write x=2
x=2 stored in cache
Attempt 2: write back

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache

CPU0: write x=1
x=1 stored in cache

CPU1: write x=2
x=2 stored in cache

CPU1: writeback
x=2 stored in memory
Attempt 2: write back

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache

CPU0: write x=1
x=1 stored in cache

CPU1: write x=2
x=2 stored in cache

CPU1: writeback
x=2 stored in memory

CPU0: writeback
x=1 stored in memory
Attempt 2: write back

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache

CPU0: write x=1
x=1 stored in cache

CPU1: write x=2
x=2 stored in cache

CPU1: writeback
x=2 stored in memory

CPU0: writeback
x=1 stored in memory

Later store x=2 lost!
Both examples violate SWMR!

**Problem 1**
CPU1 used stale value that had already been modified by CPU0.
- Solution: Invalidate copies before write proceeds!

**Problem 2**
Incorrect write-back order of modified cache lines.
- Solution: Disallow more than one modified copy!
Coherency Protocol Design Space

- **Snooping-based vs. Directory-based**
Coherency Protocol Design Space

- **Snooping-based vs. Directory-based**

Diagram showing the design space for coherency protocols in distributed operating systems. The diagram illustrates the comparison between snooping-based and directory-based approaches, with nodes representing CPUs (CPU0, CPU1), levels of cache (L1, L2), memory, and a directory.
Coherency Protocol Design Space

• **Snooping-based**
  - All coherency related traffic broadcasted to all CPUs
  - Each processor snoops and acts accordingly:
    • Invalidate lines written by other CPUs
    • Signal sharing for lines currently in cache
  - Straightforward for bus-based systems
  - Suited for small-scale systems

• **Directory-based**
  - Uses central directory to track cache line owner
  - Update copies in other caches
    • Can update all CPUs at once
      (less traffic for alternating reads and writes)
    • Multiple writes need multiple updates
      (more traffic for subsequent writes)
  - Suited for large-scale systems
• **Invalidation-based**
  - Only write misses hit bus (suited for WB caches)
  - Subsequent writes are write hits
  - Good for multiple writes to same cache line by same CPU

• **Update-based**
  - All shares of a cache line continue to hit in the cache after a write by one CPU
  - Updates have to be propagated between the individual CPUs

• Hybrid forms are possible!
A Basic Coherency Protocol: MSI

- **Modified (M)**
  - No copies on other caches; local copy modified
  - Memory is stale
- **Shared (S)**
  - Unmodified copies in one or more caches
  - Memory is up-to-date
- **Invalid (I)**
  - Not in cache

States tracked from the view of the cache controller. Sees events from:
- Local processor → processor transactions
- Other processors → snoop transactions
MSI: Processor Transitions

- **State is I, CPU reads (PrRd)**
  - Generate bus read request (BusRd)
  - Go to S
- **State is S or M, CPU reads (PrRd)**
  - No transition
- **State is S, CPU writes (PrWr)**
  - Upgrade cache line for exclusive ownership (BusRdX)
  - Go to M
- **State is M, CPU writes (PrWr)**
  - No transition
• Receiving a read snoop (BusRd) for a cache line
  - If M, write cache line back to memory (WB), transition to S
  - If S, no transition
• Receiving a exclusive ownership snoop (BusRdX)
  - If M, write cache line back to memory (WB), discard it, transition to I
  - If S, discard cache line, transition to I
MSI State Transitions

M

S

I

TU Dresden Distributed Operating Systems
MSI State Transitions

PrWr → BusRdX

PrWr → BusRdX

PrRd → BusRd

PrRd

PrRd → BusRd

PrWr → BusRdX

PrWr

PrRd

PrWr → BusRdX
MSI State Transitions

- PrWr → BusRdX
- PrWr → BusRdX
- BusRd → WB
- BusRdX → WB
- PrRd → BusRd
- PrRd → BusRd
- BusRdX
- PrWr → BusRdX
- PrWr → BusRdX

TU Dresden  Distributed Operating Systems
A common usecase is to:
- read variable A: S
- Modify A: BusRdX sent, S → M

Invalidation message pointless, if no other cache holds A.

Solved by adding Exclusive (E) state → MESI protocol
- No copies exist in other caches
- Memory is up-to-date

Variants of MESI are used by most popular processors.
MESI State Transitions

The diagram illustrates the state transitions in a MESI protocol. The states are labeled as E (Exclusive), M (Modified), I (Invalid), and S (Share). The transitions are as follows:

- PrWr → PrWr
- PrWr → BusRdX
- BusRd → WB
- BusRd → HIT
- PrRd → BusRd (HIT)
- PrRd → BusRd (!HIT)
- PrRd → BusRdX
- PrRd → BusRdX
- BusRdX → WB
- BusRdX → WB

The transitions for snooping and processor are indicated by different line styles.
MOESI: Adding Owned to MESI

• Similar to MESI, with some extensions
• Cache-to-Cache transfers of modified cache lines
  - Modified cache lines not written back to memory, but supplied to other CPUs on BusRd
  - CPU that had initial modified copy becomes “owner”
• Avoids writeback to memory when another CPU accesses cache line
  - Beneficial when cache-to-cache latency/bandwidth is better than cache-to-memory latency/bandwidth
MOESI State Transitions

TU Dresden
Distributed Operating Systems
• Bus only connected to last-level cache (e.g. L2)
  - Snoop requests are relevant to inner-level caches (e.g. L1)
  - Modifications in L1 may not be visible to L2 (and the bus)
• Idea: L2 forwards filtered transactions for L1:
  - On BusRd check if line is M/O in L1 (may be S or E in L2)
  - On BusRdX, send invalidate to L1
• Only easy for inclusive caches!

• Inclusion property
  Outer cache contains a superset of the content of its inner caches.
Memory Consistency
Concurrent programs

global variables: int i = 0;
               int k = 0;

i = 1;
if (i > 1) k = 3;  ||  i = i + 1;
if (k == 0) k = 4;
Concurrent programs

global variables:

\[
\begin{align*}
\text{int } i &= 0; \\
\text{int } k &= 0;
\end{align*}
\]

\[
\begin{align*}
i &= 1; \\
\text{if } (i > 1) & \text{ then } k = 3; \\
i &= i + 1; \\
\text{if } (k == 0) & \text{ then } k = 4;
\end{align*}
\]

\[
\begin{align*}
\text{mov } $1, [%i] \\
\text{cmp } [%i], $1 \\
\text{jgt } & \text{end} \\
\text{mov } $3, [%k] \\
\text{end:}
\end{align*}
\]

\[
\begin{align*}
\text{inc } [%i] \\
\text{cmp } [%k], $0 \\
\text{jne } & \text{end} \\
\text{mov } $4, [%k] \\
\text{end:}
\end{align*}
\]
Concurrent programs

global variables:

\[ \text{int } i = 0; \]
\[ \text{int } k = 0; \]

\[ i = 1; \]
\[ \text{if } (i > 1) \text{ k} = 3; \]
\[ i = i + 1; \]
\[ \text{if } (k == 0) \text{ k} = 4; \]

\[
\begin{align*}
\text{mov } &\$1, [\%i] \\
\text{cmp } &\%i, \$1 \\
\text{jgt } &\text{ end} \\
\text{mov } &\$3, [\%k] \\
\text{end:} \\
\text{lock; } &\text{inc } \%i \\
\text{cmp } &\%k, \$0 \\
\text{jne } &\text{ end} \\
\text{mov } &\$4, [\%k] \\
\text{end:}
\end{align*}
\]
Memory Consistency Model
defines correct shared memory behavior in terms of loads and stores in terms of how operations to different memory locations may become visible with respect to each other.

Different memory consistency models exist
- Complex models can expose more performance
- Some platforms support multiple models (SPARC)

Terminology
- **Program Order** (of a processor's operations)
  Per-processor order of memory accesses determined by the program (software)
- **Visibility Order** (of all operations)
  Order of memory accesses observed by one or more processors.
“The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called \textit{sequentially consistent}.” [Lamport 1979]

- **Program Order Requirement**
  - Each CPU issues memory operations in program order.
- **Atomicity Requirement**
  - Memory services operations one at a time
  - Memory operations appear to execute atomically with respect to other memory operations
- **Implemented by MIPS R10k**
Examples for Sequential Consistency

**CPU0**  
\[ [A] = 1; \] (a1)  
\[ [B] = 1; \] (b1)

**CPU1**  
\[ u = [B]; \] (a2)  
\[ v = [A]; \] (b2)

[A] [B] Memory  
u, v Registers
Examples for Sequential Consistency

**CPU0**

[A] = 1; (a1)
[B] = 1; (b1)

(u,v) = (1,1)

- Sequentially consistent: a1, b1, a2, b2

**CPU1**

u = [B]; (a2)
v = [A]; (b2)

[A] [B] Memory
u, v Registers

TU Dresden
Distributed Operating Systems
### Examples for Sequential Consistency

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>([A] = 1); (a1)</td>
<td>(u = [B]); (a2)</td>
</tr>
<tr>
<td>([B] = 1); (b1)</td>
<td>(v = [A]); (b2)</td>
</tr>
</tbody>
</table>

\((u,v) = (1,1)\)
- Sequentially consistent: a1, b1, a2, b2

\((u,v) = (1,0)\)
- Not sequentially consistent: b1, a2, b2, a1
- Violates program order for CPU0 (or 1)
Examples for Sequential Consistency

**CPU0**

- \([A] = 1; \) (a1)
- \(u = [B]; \) (b1)

**CPU1**

- \([B] = 1; \) (a2)
- \(v = [A]; \) (b2)

**[A] [B] Memory**

**u, v Registers**
### Examples for Sequential Consistency

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>[A] [B] Memory</th>
<th>[A] [B] Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [A] = 1; (a1) )</td>
<td>( [B] = 1; (a2) )</td>
<td>( u, v ) Registers</td>
<td>( (u,v) = (1,1) )</td>
</tr>
<tr>
<td>( u = [B]; (b1) )</td>
<td>( v = [A]; (b2) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sequentially consistent: a1, a2, b1, b2
### Examples for Sequential Consistency

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A] = 1; (a1)</td>
<td>[B] = 1; (a2)</td>
</tr>
<tr>
<td>u = [B]; (b1)</td>
<td>v = [A]; (b2)</td>
</tr>
<tr>
<td></td>
<td>[A] [B] Memory</td>
</tr>
<tr>
<td></td>
<td>u, v Registers</td>
</tr>
</tbody>
</table>

(u,v) = (1,1)
- Sequentially consistent: a1, a2, b1, b2

(u,v) = (0,0)
- Not sequentially consistent: b1, b2, a1, a2
- Violates program order for CPU0/1
In-order memory operations in SC:
- Read→Read
- Read→Write
- Write→Read
- Write→Write

Describes which program order relations hold in the visibility order of memory operations.

Weaker models relax some or all of these orderings.
Relaxing Write→Read (later reads can bypass earlier writes)
- Write followed by a read can execute out-of-order
- Typical hardware usage: Store Buffer
  - Writes must wait for cache line ownership
  - Reads can bypass writes in the buffer
  - Hides write latency

Relaxing Write→Write (later writes can bypass earlier writes)
- Write followed by a write can execute out-of-order
- Typical hardware usage: Coalescing store buffer
SB optimizes writes to memory and/or caches to optimize interconnect accesses.

CPU can continue before write is completed.

**Store forwarding** allows reads from local CPU to see pending writes in the SB.

SB invisible to remote CPUs.

FIFO vs. non-FIFO. Writes can be combined, may reorder writes on some architectures.
• In-order memory operations:
  - Read→Read
  - Read→Write
  - Write→Write

• Out-of-order memory operations:
  - Write→Read (later reads can bypass earlier writes)
    • Unless both to same location
    • Breaks Dekker's algorithm for mutual exclusion
  - Write→Read to same location must execute in-order
    • No forwarding from the store buffer
bool flag[2] = {false, false}; // Intention to enter
int turn = 0; // Who's next?

CPU0

P: flag[0] = true;
while (flag[1]) {
    if (turn == 1) {
        flag[0] = false;
        goto P;
    }
}
// Critical section
flag[0] = false;
turn = 1;

CPU1

P: flag[1] = true;
while (flag[0]) {
    if (turn == 0) {
        flag[1] = false;
        goto P;
    }
}
// Critical section
flag[1] = false;
turn = 0;
Dekker's Algorithm on z Series

bool flag[2] = {false,false}; // Intention to enter
int turn = 0; // Who's next?

CPU0

P: flag[0] = true;
while (flag[1]) {
    If (turn == 1) {
        flag[0] = false;
        goto P;
    }
}
// Critical section
flag[0] = false;
turn = 1;

CPU1

Buffered P: flag[1] = true;
while (flag[0]) {
    If (turn == 0) {
        flag[1] = false;
        goto P;
    }
}
// Critical section
flag[1] = false;
turn = 0;
In-order memory operations:
- Read→Read
- Read→Write
- Write→Write

Out-of-order memory operations:
- Write→Read (later reads can bypass earlier writes)
  - Forwarding of pending writes in the store buffer to successive reads to the same location
  - Store buffer is FIFO
  - Breaks Peterson's algorithm for mutual exclusion
bool flag[2] = {false, false}; // Intention to enter
int turn = 0;              // Who's next?

CPU0

flag[0] = true;
turn = 1;
while (turn == 1 && flag[1]) {}
// Critical section
flag[0] = false;

CPU1

flag[1] = true;
turn = 0;
while (turn == 0 && flag[0]) {}
// Critical section
flag[1] = false;
Peterson's Algorithm on TSO

```c
bool flag[2] = {false, false};  // Intention to enter
int turn = 0;                   // Who's next?

CPU0
flag[0] = true;
turn = 1;
while (turn == 1 && flag[1]) {} // Critical section
flag[0] = false;

CPU1
Buffered flag[1] = true;
turn = 0;
while (turn == 0 && flag[0]) {} // Critical section
flag[1] = false;
```
bool flag[2] = {false,false}; // Intention to enter
int turn = 0;               // Who's next?

CPU0
flag[0] = true;
turn = 1;
while (turn == 1 && flag[1]) {}
// Critical section
flag[0] = false;

CPU1
flag[1] = true;
turn = 0;
while (turn == 0 && flag[0]) {}
// Critical section
flag[1] = false;

Loading turn orders accesses on zSeries, but not on TSO!
CPU0
[A] = 1; (a1)
u = [A]; (b1)
w = [B]; (c1)

CPU1
[B] = 1; (a2)
v = [B]; (b2)
x = [A]; (c2)

• (u,v,w,x) = (1,1,0,0)
  - Not possible with SC and z Series
  - Possible with TSO
**CPU0**

\[
\begin{align*}
[A] &= 1; \quad (a1) \\
u &= [A]; \quad (b1) \\
w &= [B]; \quad (c1)
\end{align*}
\]

**CPU1**

\[
\begin{align*}
[B] &= 1; \quad (a2) \\
v &= [B]; \quad (b2) \\
x &= [A]; \quad (c2)
\end{align*}
\]

- \((u,v,w,x) = (1,1,0,0)\)
  - Not possible with SC and z Series
  - Possible with TSO
    - \(b1, b2, c1, c2, a1, a2\)
    - \(b1\) reads \([A]\) from write buffer
Processor Consistency (PC)

- Similar to Total Store Order (TSO)
- Additionally supports multiple cached memory copies
  - Relaxed atomicity for write operations
    - Each write broken into suboperations to update cached copies of other CPUs
  - Non-unique write order: **per-CPU visibility order**

- Additional coherency requirement
  - All write suboperations to the same location complete in the same order across all memory copies (or in other words: each processor sees writes to the same location in the same order)
**CPU0**

\[ [A] = 1; \text{(a1)} \]

**CPU1**

\[ u = [A]; \text{(a2)} \]

\[ [B] = 1; \text{(b2)} \]

**CPU2**

\[ v = [B]; \text{(a3)} \]

\[ w = [A]; \text{(b3)} \]

- \((u,v,w) = (1,1,0)\)
  - Not possible with SC, z Series, TSO
  - Possible with Processor Consistency (PC)
### PC vs. SC, z Series, TSO

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>CPU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>([A] = 1); (a1)</td>
<td>(u = [A]); (a2)</td>
<td>(v = [B]); (a3)</td>
</tr>
<tr>
<td>([B] = 1); (b2)</td>
<td>(w = [A]); (b3)</td>
<td></td>
</tr>
</tbody>
</table>

- \((u,v,w) = (1,1,0)\)
  - Not possible with SC, z Series, TSO
  - Possible with Processor Consistency (PC)
    - CPU0 sets \([A]\), sends update to other CPUs
    - CPU1 gets update, sets \([B]\), sends update
    - CPU2 sees update from CPU1, but hasn't seen update from CPU0 yet
  - Single memory bus enforces single visibility order

TU Dresden

Distributed Operating Systems
Causality

CPU0
[A] = 1;
[B] = 1;

CPU1
while ([A] == 0);
[B] = 1;

CPU2
while ([B] == 0);
print [A];

Write Atomicity
All cores see writes at the same time (and the same order).

Relaxing write atomicity
- CPU0 writes [A]; sends update to CPU1/2
- CPU1 receives; writes [B]; sends update to CPU2
- CPU2 receives update from CPU1, prints [A] = 0
- CPU2 receives update from CPU0
Not sequentially consistent!
• In-order memory operations:
  - Read→Read
  - Read→Write

• Out-of-order memory operations:
  - Write→Read (later reads can bypass earlier writes)
    • Forwarding of pending writes to successive reads to the same location
  - Write→Write (later writes can bypass earlier writes)
    • Unless both are to the same location
    • Breaks naive producer-consumer code

• Write atomicity is maintained → single visibility order
CPU0
\[ [A] = 1; \quad (a1) \]
\[ [B] = 1; \quad (b1) \]
\[ [\text{Flag}] = 1; \quad (c1) \]

CPU1
\[ \text{while} \quad ([\text{Flag}] == 0); \quad (a2) \]
\[ u = [A]; \quad (b2) \]
\[ v = [B]; \quad (c2) \]

• \((u,v) = (0,0)\) or \((0,1)\) or \((1,0)\)
  - Not possible with SC, z Series, TSO, PC
  - Possible with PSO
CPU0 | CPU1
-----|-----
[A] = 1; (a1) | while ([Flag] == 0); (a2)
[B] = 1; (b1) | u = [A]; (b2)
[Flag] = 1; (c1) | v = [B]; (c2)

• (u,v) = (0,0) or (0,1) or (1,0)
  - Not possible with SC, z Series, TSO, PC
  - Possible with PSO
    • c1,a2,b2,c2,a1,b1
In addition to previous relaxations:
- Read→Read (later reads can bypass earlier reads)
  - Read followed by read can execute out-of-order
- Read→Write (later writes can bypass earlier reads)
  - Read followed by a write can execute out-of-order

Examples
- Weak Ordering (WO)
- Release Consistency (RC)
- DEC Alpha
- SPARC V9 Relaxed Memory Model (RMO)
- PowerPC
- Itanium (IA-64)
Distributed Operating Systems

Release Consistency (RC)

• Distinguishes memory operations as
  - Ordinary (data)
  - Special
    • Sync (synchronization)
    • Nsync (asynchronous data)
• Sync operations classified as
  - Acquire
    • Read operation for gaining access to a shared resource
    • e.g., spinning on a flag to be set, reading a pointer
  - Release
    • Write operation for granting permission to a shared resource
    • e.g., setting a synchronization flag
Flavors of Release Consistency

- **RC\textsubscript{SC}**
  - Sequential consistency between special operations
  - Program order enforced between:
    - acquire $\rightarrow$ all
    - all $\rightarrow$ release
    - special $\rightarrow$ special

- **RC\textsubscript{PC}**
  - Processor consistency between special operations
  - Program order enforced between:
    - acquire $\rightarrow$ all
    - all $\rightarrow$ release
    - special $\rightarrow$ special, except release followed by acquire
Enforcing Ordering: Synchronization Instructions

- x86-32bit (IA32)
  - lfence, sfence, mfence (load, store, memory fence)
- Alpha
  - mb (memory barrier), wmb (write memory barrier)
- SPARC (PSO)
  - stbar (store barrier)
- SPARC (RMO)
  - membar (4-bit encoding for r-r, r-w, w-r, w-w)
- PowerPC
  - sync (similar to Alpha mb, except r-r), lwsync
  - eieio (enforce in-order execution of I/O)
Standardized memory models for HLL:
- C / C++ 2011
- Java

Basic model: Sequential Consistency for data-race free programs (SC-DRF)

A data-race free program will execute sequentially consistent.

**Data Race (informal)**
Multiple threads access a memory location without synchronization, one of them is a writer.
\begin{itemize}
  \item \textbf{Thread 1} \hspace{1cm} \textbf{Thread 2}
  \begin{align*}
  \text{a = b = 0;} \\
  \text{mtx\_lock(l);} \quad \text{x = a;}
  \text{a = 1;} \\
  \text{b = 1;} \quad \text{y = b;}
  \text{mtx\_unlock(l);} \\
  \end{align*}
\end{itemize}

\textbf{Not data-race free:}
\begin{itemize}
  \item a,b accessed without synchronization
  \item (x,y) = (0,0) (1,0) (0,1) (1,1) all legal!
  \item Need to add synchronization to Thread 2
\end{itemize}
a = b = 0;

**Thread 1**
- `mtx_lock(l);`
- `a = 1;`
- `b = 1;`
- `mtx_unlock(l);`

**Thread 2**
- `x = a;`
- `y = b;`

Not data-race free:
- `a, b` accessed without synchronization
- `(x, y) = (0,0) (1,0) (0,1) (1,1)` all legal!
- Need to add synchronization to Thread 2

With synchronization yields either `(0,0)` or `(1,1)`
Enforcing Memory Ordering in C++

- Mutexes may cause scalability issues
- C++ 11 offers rich set of atomic memory operations (std::atomic)
  - Implements RC:
    - Atomic reads acquire
    - Atomic stores release
  - Can use weaker ordering if desired
  - Compare-and-Swap
  - Add/Sub/And/Or/Xor/...

- Does the right thing on all platforms
  - Adds appropriate memory barriers
  - Uses locked instructions as necessary
  - May use locks on certain platforms!
A Primer on Memory Consistency and Cache Coherence
Sorin, Hill, Wood; 2011

atomic<> Weapons: The C++ Memory Model and Modern Hardware (Video)
Sutter; 2013

Shared memory consistency models: a tutorial
Adve, Gharachorloo; 1996

IA Memory Model
Richard Hudson; Google Tech Talk 2008

Memory Ordering in Modern Microprocessors
McKenney; Linux Journal 2005

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs
Lamport, 1979

PowerPC Storage Model