Script generated by TTT

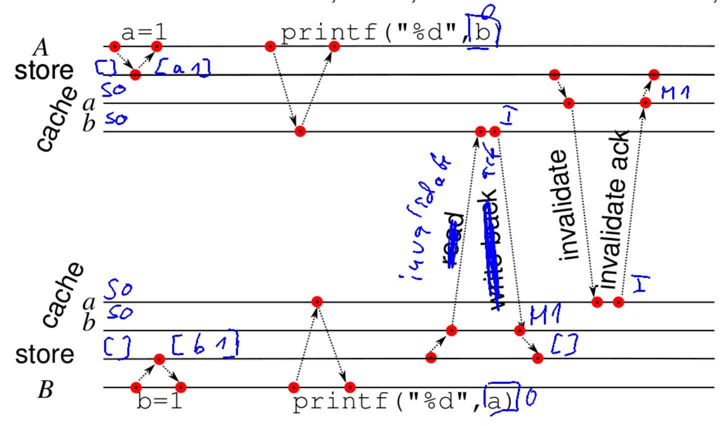
Title:	Petter: Programmiersprachenh (31.10.2018)
Date:	Wed Oct 31 14:13:27 CET 2018
Duration:	78:48 min
Pages:	37

Happened-Before Model for TSO

Thread A	Thread B

`a = 1;`
`printf("%d",b);`
	`b = 1;`
	`printf("%d",a);`

Assume cache A contains: a: S0, b: S0, cache B contains: a: S0, b: S0



TSO in the Wild: x86

The x86 CPUs, powering desktops and servers around the world is a common representative of a TSO Memory Model based CPU.

- FIFO store buffers keep quite strong consistency properties
- The major obstacle to Sequential Consistency is

$$\boxed{\mathtt{St}_i[a] \leq \mathtt{Ld}_i[b] \quad \not\Rightarrow \quad \mathtt{St}_i[a] \sqsubseteq \mathtt{Ld}_i[b]}$$

  - modern x86 CPUs provide the $\boxed{\texttt{mfence}}$ instruction
  - `mfence` orders all memory instructions:

$$\mathtt{Op}_i \boxed{\leq \mathit{mfence}()} \leq \mathtt{Op}_i' \quad \Rightarrow \quad \mathtt{Op}_i \sqsubseteq \mathtt{Op}_i'$$

- a fence between write and loads gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)

⇝ use fences only when necessary

PSO Model: Formal Spec [SI92]

Definition (Partial Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total
$$\forall_{a,b \,\in\, addr\; i,j \,\in CPU} \quad (\mathtt{St}_i[a] \sqsubseteq \mathtt{St}_j[b]) \vee (\mathtt{St}_j[b] \sqsubseteq \mathtt{St}_i[a])$$
2. Fenced stores in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\boxed{\mathtt{St}_i[a] \leq \mathtt{sfence()} \leq \mathtt{St}_i[b] \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]}$$
3. Stores to the same address in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\boxed{\mathtt{St}_i[a] \leq \mathtt{St}_i[a]' \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[a]'}$$
4. Loads preceding an other operation (wrt. program order $\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{Ld}_i[a] \leq \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$
5. A load's value is determined by the latest write as observed by the local CPU
$$\mathit{val}(\mathtt{Ld}_i[a]) = \mathit{val}(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq} (\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \leq \mathtt{Ld}_i[a]\}))$$

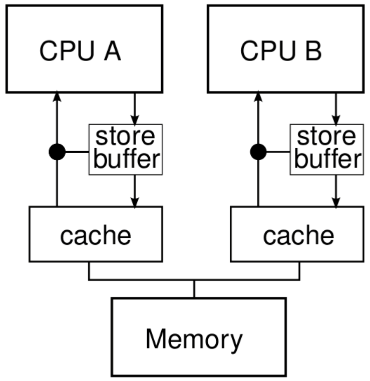
⚠ Now also stores are not guaranteed to be in order any more:

$$\mathtt{St}_i[a] \leq \mathtt{St}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

⇝ What about sequential consistency for the whole system?

Store Buffers

⚠ *Abstract Machine Model:* defines semantics of memory accesses



- put *each* store into a *store buffer* and continue execution
- Store buffers apply stores in various orders:
  - FIFO (Sparc/x86-*TSO*)
  - unordered (Sparc *PSO*)
- ⚠ program order still needs to be observed locally
  - store buffer snoops read channel and
  - on matching address, returns the youngest value in buffer

PSO Model: Formal Spec [SI92]

Definition (Partial Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total
$$\forall_{a,b \in addr\ i,j \in CPU} \quad (\mathtt{St}_i[a] \sqsubseteq \mathtt{St}_j[b]) \vee (\mathtt{St}_j[b] \sqsubseteq \mathtt{St}_i[a])$$
2. Fenced stores in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{St}_i[a] \leq \mathtt{sfence()} \leq \mathtt{St}_i[b] \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$
3. Stores to the same address in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{St}_i[a] \leq \mathtt{St}_i[a]' \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[a]'$$
4. Loads preceding an other operation (wrt. program order $\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{Ld}_i[a] \leq \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$
5. A load's value is determined by the latest write as observed by the local CPU
$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq} (\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \leq \mathtt{Ld}_i[a]\}))$$

⚠ Now also stores are not guaranteed to be in order any more:
$$\mathtt{St}_i[a] \leq \mathtt{St}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

$\rightsquigarrow$ What about sequential consistency for the whole system?

Happened-Before Model for PSO

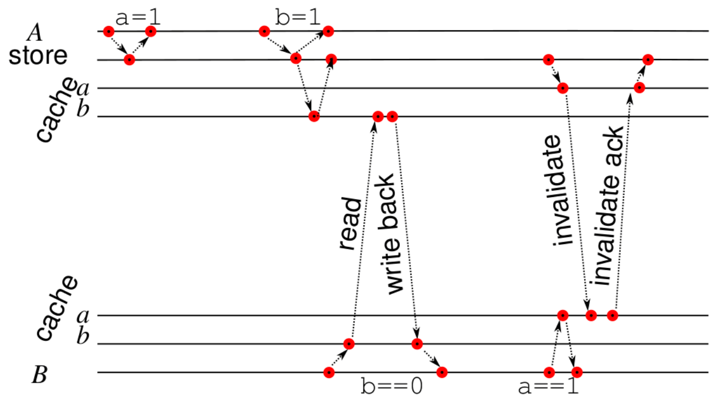
Thread A

```
a = 1;
b = 1;
```

Thread B

```
while (b == 0) {};
assert(a == 1);
```

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I



Explicit Synchronization: Write Barrier

Overtaking of messages *may be desirable* and does not need to be prohibited in general.

- generalized store buffers render programs incorrect that assume sequential consistency between *different* CPUs
- whenever a store in front of another operation in one CPU must be observable in this order *by a different CPU*, an explicit *write barrier* has to be inserted
  - a write barrier marks all current store operations in the store buffer
  - the next store operation is only executed when all marked stores in the buffer have completed

Happened-Before Model for Write Barriers

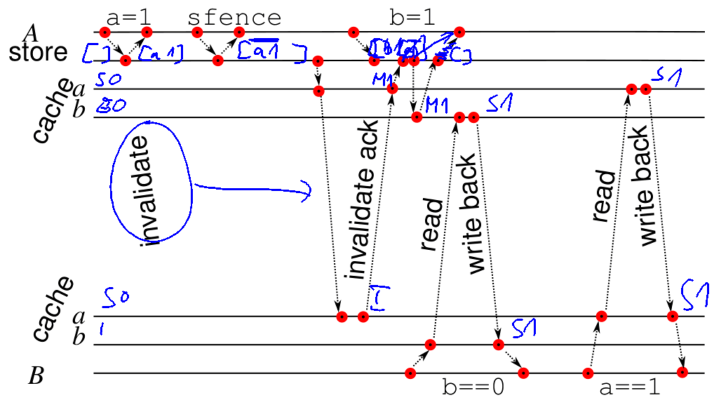
Thread A

```
a = 1;
sfence();
b = 1;
```

Thread B

```
while (b == 0) {};
assert(a == 1);
```

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I



Further weakening the model: O-o-O Reads

PSO Model: Formal Spec [SI92]

Definition (Partial Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total

$$\forall_{a,b\ \in\ addr\ i,j\ \in CPU}\quad (\mathtt{St}_i[a] \sqsubseteq \mathtt{St}_j[b]) \vee (\mathtt{St}_j[b] \sqsubseteq \mathtt{St}_i[a])$$

2. Fenced stores in program order ($\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{St}_i[a] \le \mathtt{sfence()} \le \mathtt{St}_i[b] \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

3. Stores to the same address in program order ($\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{St}_i[a] \le \mathtt{St}_i[a]' \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[a]'$$

4. Loads preceding an other operation (wrt. program order $\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{Ld}_i[a] \le \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$

5. A load's value is determined by the latest write as observed by the local CPU

$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq} (\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \le \mathtt{Ld}_i[a]\}))$$

⚠ Now also stores are not guaranteed to be in order any more:

$$\mathtt{St}_i[a] \le \mathtt{St}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

$\leadsto$ What about sequential consistency for the whole system?

TSO Model: Formal Spec [SI92]

Definition (Total Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total

$$\forall_{a,b\ \in\ addr\ i,j\ \in CPU}\quad (\mathtt{St}_i[a] \sqsubseteq \mathtt{St}_j[b]) \vee (\mathtt{St}_j[b] \sqsubseteq \mathtt{St}_i[a])$$

2. Stores in program order ($\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{St}_i[a] \le \mathtt{St}_i[b] \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

3. Loads preceding an other operation (wrt. program order $\le$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{Ld}_i[a] \le \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$

4. A load's value is determined by the latest write as observed by the local CPU

$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq} (\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \le \mathtt{Ld}_i[a]\}))$$

Particularly, one ordering property is not guaranteed:

$$\mathtt{St}_i[a] \le \mathtt{Ld}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{Ld}_i[b]$$

⚠ Local stores may be observed earlier by local loads then from somewhere else!

TSO in the Wild: x86

The x86 CPUs, powering desktops and servers around the world is a common representative of a TSO Memory Model based CPU.

- FIFO store buffers keep quite strong consistency properties
- The major obstacle to Sequential Consistency is

$$\texttt{St}_i[a] \leq \texttt{Ld}_i[b] \quad \not\Rightarrow \quad \texttt{St}_i[a] \sqsubseteq \texttt{Ld}_i[b]$$

  - modern x86 CPUs provide the `mfence` instruction
  - `mfence` orders all memory instructions:

$$\texttt{Op}_i \leq \mathit{mfence}() \leq \texttt{Op}_i' \quad \Rightarrow \quad \texttt{Op}_i \sqsubseteq \texttt{Op}_i'$$

- a fence between write and loads gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)

⇝ use fences only when necessary

PSO Model: Formal Spec [Sl92]

Definition (Partial Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total
$$\forall_{a,b\,\in\,addr\;i,j\,\in CPU} \quad (\texttt{St}_i[a] \sqsubseteq \texttt{St}_j[b]) \vee (\texttt{St}_j[b] \sqsubseteq \texttt{St}_i[a])$$
2. Fenced stores in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\texttt{St}_i[a] \boxed{\leq \texttt{sfence()} \leq} \texttt{St}_i[b] \Rightarrow \texttt{St}_i[a] \sqsubseteq \texttt{St}_i[b]$$
3. Stores to the same address in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\texttt{St}_i[a] \leq \texttt{St}_i[a]' \Rightarrow \texttt{St}_i[a] \sqsubseteq \texttt{St}_i[a]'$$
4. Loads preceding an other operation (wrt. program order $\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\texttt{Ld}_i[a] \leq \texttt{Op}_i[b] \Rightarrow \texttt{Ld}_i[a] \sqsubseteq \texttt{Op}_i[b]$$
5. A load's value is determined by the latest write as observed by the local CPU
$$val(\texttt{Ld}_i[a]) = val(\texttt{St}_j[a] \mid \texttt{St}_j[a] = \max_{\sqsubseteq}(\{\texttt{St}_k[a] \mid \texttt{St}_k[a] \sqsubseteq \texttt{Ld}_i[a]\} \cup \{\texttt{St}_i[a] \mid \texttt{St}_i[a] \leq \texttt{Ld}_i[a]\}))$$

⚠ Now also stores are not guaranteed to be in order any more:

$$\texttt{St}_i[a] \leq \texttt{St}_i[b] \not\Rightarrow \texttt{St}_i[a] \sqsubseteq \texttt{St}_i[b]$$

⇝ What about sequential consistency for the whole system?

Happened-Before Model for Write Barriers

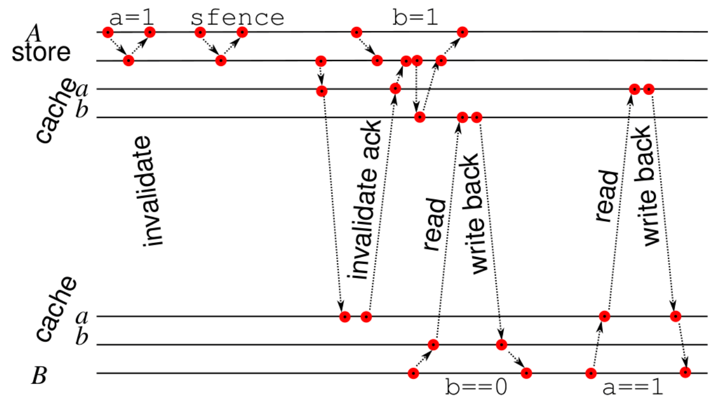
Thread A

```
a = 1;
sfence();
b = 1;
```

Thread B

```
while (b == 0) {};
assert(a == 1);
```

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I

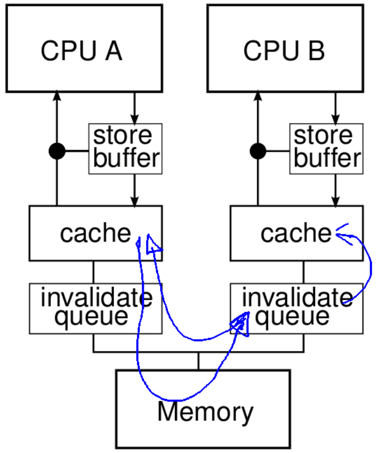


Further weakening the model: O-o-O Reads

Relaxed Memory Order

Communication of cache updates is still costly:

- a cache-intense computation can fill up store buffers in CPUs
- ⇝ waiting for invalidation acknoledgements may still happen
- invalidation acknoledgements are delayed on busy caches



⇝ immediately acknowledge an invalidation and apply it later

- put each invalidate message into an *invalidate queue*
- if a *MESI message* needs to be sent regarding a cache line in the invalidate queue then wait until the line is invalidated

⚠ local loads and stores do *not* consult the invalidate queue

⇝ What about sequential consistency?

RMO Model: Formal Spec [SI94]

Definition (Relaxed Memory Order)

1. Fenced memory accesses in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\texttt{Op}_i[a] \leq \texttt{mfence()} \leq \texttt{Op}_i[b] \Rightarrow \texttt{Op}_i[a] \sqsubseteq \texttt{Op}_i[b]$$
2. Stores to the same address in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\texttt{St}_i[a] \leq \texttt{St}_i[a]' \Rightarrow \texttt{St}_i[a] \sqsubseteq \texttt{St}_i[a]'$$
3. Operations dependent on a load (wrt. *dependence* $\rightarrow$) are embedded in the memory order ($\sqsubseteq$)
$$\texttt{Ld}_i[a] \rightarrow \texttt{Op}_i[b] \Rightarrow \texttt{Ld}_i[a] \sqsubseteq \texttt{Op}_i[b]$$
4. A load's value is determined by the latest write as observed by the local CPU
$$val(\texttt{Ld}_i[a]) = val(\texttt{St}_j[a] \mid \texttt{St}_j[a] = \max_{\sqsubseteq} (\{\texttt{St}_k[a] \mid \texttt{St}_k[a] \sqsubseteq \texttt{Ld}_i[a]\} \cup \{\texttt{St}_i[a] \mid \texttt{St}_i[a] \leq \texttt{Ld}_i[a]\}))$$

⚠ Now we need the notion of *dependence* $\rightarrow$:

- Memory access to the same address:
$$\texttt{St}_i[a] \leq \texttt{Ld}_i[a] \quad \Rightarrow \quad \texttt{St}_i[a] \rightarrow \texttt{Ld}_i[a]$$
- Register reads are dependent on latest register writes:
$$\texttt{Op}_i[a]'' = \max_{\leq} (\texttt{Op}_i[a]' \mid targetreg(\texttt{Op}_i[a]') = srcreg(\texttt{Op}_i[b]) \wedge \texttt{Op}_i[a]' \leq \texttt{Op}_i[b]) \quad \Rightarrow \quad \texttt{Op}_i[a]'' \rightarrow \texttt{Op}_i[b]$$
- Stores within branched blocks are dependent on branch conditionals:
$$(\texttt{Op}_i[a] \leq \texttt{St}_i[b]) \wedge \boxed{\texttt{Op}_i[a] \rightarrow condbranch \leq \texttt{St}_i[b]} \quad \Rightarrow \quad \texttt{Op}_i[a] \rightarrow \texttt{St}_i[b]$$

Happened-Before Model for Invalidate Queues

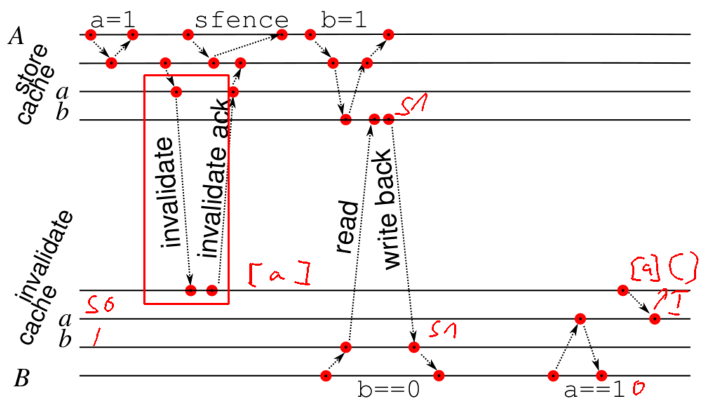
Thread A

```
a = 1;
sfence();
b = 1;
```

Thread B

```
while (b == 0) {};
assert(a == 1);
```

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I



Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.

- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads
- insert an explicit *read barrier* before the read access
  - a read barrier marks all entries in the invalidate queue
  - the next read operation is only executed once all marked invalidations have completed
- a read barrier *before* each read gives sequentially consistent read behavior (and is as slow as a system without invalidate queue)

⇝ match each write barrier in one process with a read barrier in another process

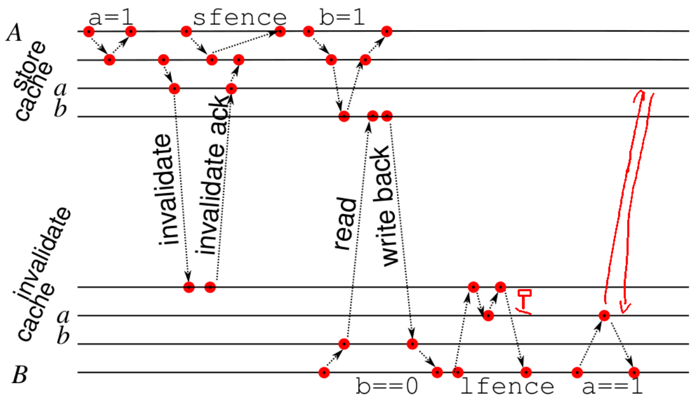
Happened-Before Model for Read Barriers

Thread A

```
a = 1;
sfence();
b = 1;
```

Thread B

```
while (b == 0) {};
lfence();
assert(a == 1);
```



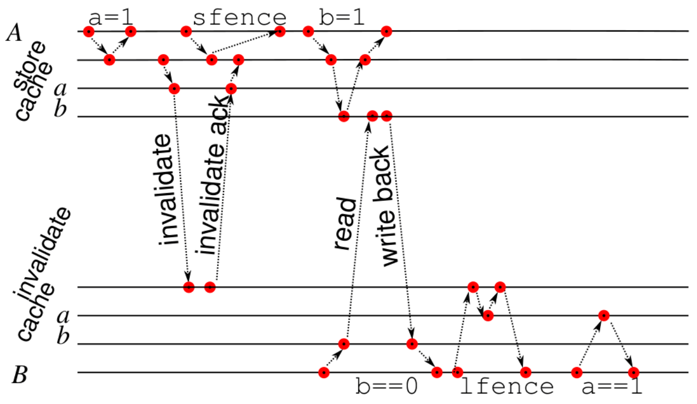
Happened-Before Model for Read Barriers

Thread A

```
a = 1;
sfence();
b = 1;
```

Thread B

```
while (b == 0) {};
lfence();
assert(a == 1);
```



Example: The Dekker Algorithm on RMO Systems

Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of *two* processes with busy waiting.

```
//flag[] is boolean array; and turn is an integer
flag[0] = false;
flag[1] = false;
turn    = 0;    // or 1
```

```
P0:
flag[0] = true;
while (flag[1] == true)
  if (turn != 0) {
     flag[0] = false;
     while (turn != 0) {
       // busy wait
     }
     flag[0] = true;
  }
// critical section
turn    = 1;
flag[0] = false;
```

Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of *two* processes with busy waiting.

```
//flag[] is boolean array; and turn is an integer
flag[0] = false;
flag[1] = false;
turn    = 0;   // or 1
```

```
P0:
flag[0] = true;
while (flag[1] == true)
  if (turn != 0) {
      flag[0] = false;
      while (turn != 0) {
        // busy wait
      }
      flag[0] = true;
  }
// critical section
turn    = 1;
flag[0] = false;
```

```
P1:
flag[1] = true;
while (flag[0] == true)
  if (turn != 1) {
      flag[1] = false;
      while (turn != 1) {
        // busy wait
      }
      flag[1] = true;
  }
// critical section
turn    = 0;
flag[1] = false;
```

The Idea Behind Dekker

Communication via three variables:

- `flag[i]==true` process P_i wants to enter its critical section
- `turn==i` process P_i has priority when both want to enter

```
P0:
flag[0] = true;
while (flag[1] == true)
  if (turn != 0) {
      flag[0] = false;
      while (turn != 0) {
        // busy wait
      }
      flag[0] = true;
  }
// critical section
turn    = 1;
flag[0] = false;
```

In process P_i:

- if P_{1-i} does not want to enter, proceed immediately to the critical section

Dekker's Algorithm and RMO

Problem: Dekker's algorithm requires sequential consistency.
Idea: insert memory barriers between all variables common to both threads.

Dekker's Algorithm and RMO

Problem: Dekker's algorithm requires sequential consistency.
Idea: insert memory barriers between all variables common to both threads.

```
P0:
flag[0] = true;
sfence();
while (lfence(), flag[1] == true)
  if (lfence(), turn != 0) {
      flag[0] = false;
      sfence();
      while (lfence(), turn != 0){
        // busy wait
      }
      flag[0] = true;
      sfence();
  }
// critical section
turn    = 1;
sfence();
flag[0] = false; sfence();
```

- insert a load memory barrier `lfence()` in front of every read from common variables

Summary: Relaxed Memory Models

Highly optimized CPUs may use a *relaxed memory model*:

- reads and writes are not synchronized unless requested by the user
- many kinds of memory barriers exist with subtle differences

$\rightsquigarrow$ ARM, PowerPC, Alpha, ia-64, even x86 ($\rightsquigarrow$ SSE Write Combining)

$\rightsquigarrow$ memory barriers are the “lowest-level” of synchronization

Discussion

Memory barriers reside at the lowest level of synchronization primitives.

Where are they useful?

- when blocking should not de-schedule threads
- when several processes implement automata and coordinate their transitions via common synchronized variables

$\rightsquigarrow$ protocol implementations

$\rightsquigarrow$ OS provides synchronization facilities based on memory barriers

Why might they not be appropriate?

- difficult to get right, best suited for specific well-understood algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck

Memory Models and Compilers

Before Optimization

```
int x = 0;
for (int i=0;i<100;i++){
    x = 1;
    printf("%d",x);
}
```

Memory Models and Compilers

Before Optimization

```
int x = 0;
for (int i=0;i<100;i++){
    x = 1;
    printf("%d",x);
}
```

After Optimization

```
int x = 1;
for (int i=0;i<100;i++){
    printf("%d",x);
}
```

Standard Program Optimizations

comprises *loop-invariant code motion* and *dead store elimination*, e.g.

Memory Models and C-Compilers

Keeping semantics I

```
int x = 0;
for (int i=0;i<100;i++){
    sfence();
    x = 1;
    printf("%d",x);
}
```

Memory Models and C-Compilers

Keeping semantics I

```
int x = 0;
for (int i=0;i<100;i++){
    sfence();
    x = 1;
    printf("%d",x);
}
```

Keeping semantics II

```
volatile int x = 0;
for (int i=0;i<100;i++){
    x = 1;
    printf("%d",x);
}
```

- Compilers may also reorder store instructions
- Write barriers keep the compiler from reordering across
- The specification of `volatile` keeps the *C-Compiler* from reordering memory accesses to this address

Summary

Learning Outcomes

1. Strict Consistency
2. Happened-before Relation
3. Sequential Consistency
4. The MESI Cache Model
5. TSO: FIFO store buffers
6. PSO: store buffers
7. RMO: invalidate queues
8. Reestablishing Sequential Consistency with memory barriers
9. Dekker's Algorithm for Mutual Exclusion

Future Many-Core Systems: NUMA

Many-Core Machines' Read Responses congest the bus

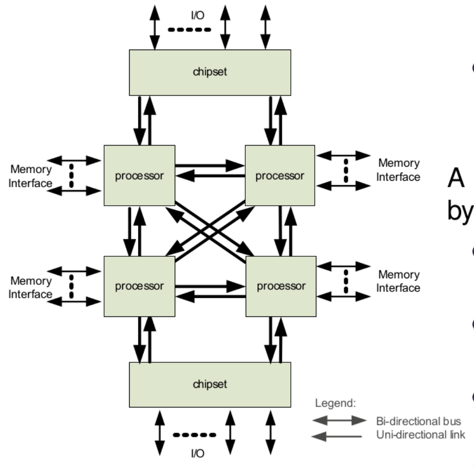
In that case: Intel's *MESIF* (Forward) to reduce communication overhead.

⚠ But in general, Symmetric multi-processing (SMP) has its limits:

- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus

Overhead of NUMA Systems

Communication overhead in a NUMA system.



source: [Int09]

- Processors in a NUMA system may be fully or partially connected.
- The directory of who stores an address is partitioned amongst processors.

A cache miss that cannot be satisfied by the local memory at A:

- A sends a retrieve request to processor B owning the directory
- B tells the processor C who holds the content
- C sends data (or status) to A and sends acknowledge to B
- B completes transmission by an acknowledge to A

References

Intel.
An introduction to the intel quickpath interconnect.
Technical Report 320412, 2009.

Leslie Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Commun. ACM, 21(7):558–565, July 1978.

Paul E. McKenny.
Memory Barriers: a Hardware View for Software Hackers.
Technical report, Linux Technology Center, IBM Beaverton, June 2010.

Mark S. Papamarcos and Janak H. Patel.
A low overhead coherence solution for multiprocessors with private cache memories.
In *In Proc. 11th ISCA*, pages 348–354, 1984.

CORPORATE SPARC International, Inc.
The SPARC Architecture Manual: Version 8.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

CORPORATE SPARC International, Inc.
The SPARC Architecture Manual (Version 9).
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.