TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Memory Consistency

Dr. Michael Petter
Winter term 2019

Thread A

```
void foo(void) {
  a = 1;
  b = 1;
}
```

Thread B

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

Intuition: the assertion will never fail

Thread A

```
void foo(void) {
  a = 1;
  b = 1;
}
```

Thread B

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

Intuition: the assertion will never fail

⚠ Real execution: given enough tries, the assertion may eventually fail

⇝ in need of defining a *Memory Model*

Memory Models

Memory interactions behave differently in presence of

- multiple concurrent threads
- data replication in hierarchical and/or distributed memory systems
- deferred communication of updates

Memory Models are a product of negotiating

- restrictions of freedom of implementation to guarantee race related properties
- establishment of freedom of implementation to enable *program* and *machine model* optimizations

$\leadsto$ Modern Languages include the memory model in their language definition

Strict Consistency

Motivated by sequential computing, we intuitively implicitely transfer our idea of semantics of memory accesses to concurrent computation. This leads to our idealistic model *Strict Consistency*:

Definition (Strict consistency)

Independently of which process reads or writes, the value from the most recent write to a location is observable by reads from the respective location immediately *after* the write occurs.

Although idealistically desired, practically not existing

⚠ absolute global time problematic

⚠ physically not possible

⇝ strict consistency is too strong to be realistic

Abandoning absolute time

Thread A

```
void foo(void) {
  a = 1;
  b = 1;
}
```

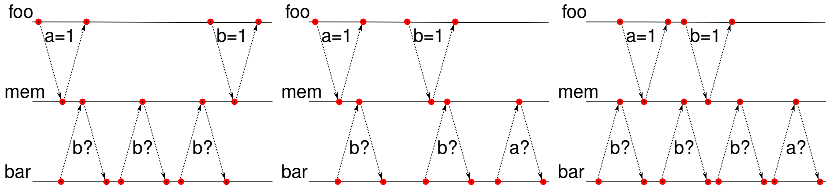
Thread B

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

- initial state of `a` and `b` is `0`
- A writes `a` before it writes `b`
- B should see `b` go to `1` before executing the `assert` statement
- the `assert` statement should always hold

⇝ here correctness means: writing a `1` to `a` *happens before* reading a `1` in `b`

Still, *any* of the following may happen:



⇝ Idea: state correctness in terms of what event *may* happen before another one

Happend-Before Relation and Diagram

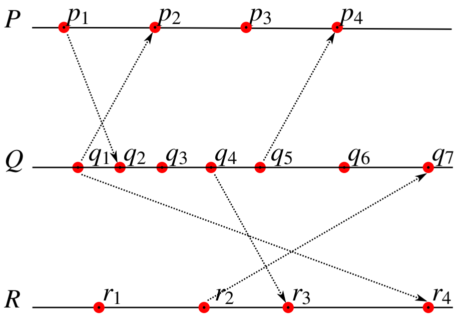
Events in a Distributed System

A process as a series of events [Lam78]: Given a distributed system of processes $P, Q, R, \ldots$, each process P consists of events ●p_1, ●$p_2, \ldots$.

Events in a Distributed System

A process as a series of events [Lam78]: Given a distributed system of processes $P, Q, R, \ldots$, each process P consists of events $\bullet p_1, \bullet p_2, \ldots$.

Example:



- event $\bullet p_i$ in process P *happened before* $\bullet p_{i+1}$
- if $\bullet p_i$ is an event that sends a message to Q then there is some event $\bullet q_j$ in Q that receives this message and $\bullet p_i$ *happened before* $\bullet q_j$

The Happened-Before Relation

Definition

If an event p *happened before* an event q then $p \rightarrow q$.

The Happened-Before Relation

Definition

If an event p *happened before* an event q then $p \rightarrow q$.

Observe:

- $\rightarrow$ is partial (neither $p \rightarrow q$ or $q \rightarrow p$ may hold)
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
- $\rightarrow$ is transitive ($p \rightarrow q \wedge q \rightarrow r$ then $p \rightarrow r$)
- $\rightarrow$ is asymmetric (if $p \rightarrow q$ then $\neg(q \rightarrow p)$)

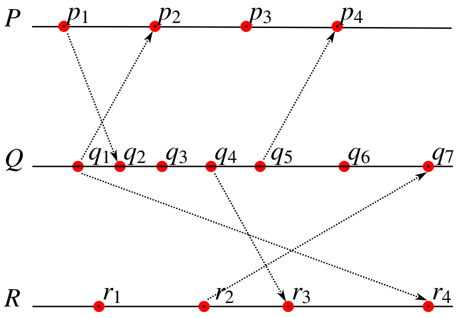
$\leadsto$ the $\rightarrow$ relation is a *strict partial order*

Concurrency in Happened-Before Diagrams

Let $a \not\rightarrow b$ abbreviate $\neg(a \rightarrow b)$.

Definition

Two distinct events p and q are said to be *concurrent* if $p \not\rightarrow q$ and $q \not\rightarrow p$.



- $p_1 \rightarrow r_4$ in the example
- p_3 and q_3 are, in fact, concurrent since $p_3 \not\rightarrow q_3$ and $q_3 \not\rightarrow p_3$

Ordering

Let C be a *logical clock* i.e.C assigns a *globally unique* time-stamp $C(p)$ to each event p.

Definition (Clock Condition)

Function C satisfies the *clock condition* **if** for any events p, q

$$p \rightarrow q \quad \Longrightarrow \quad C(p) < C(q)$$

Ordering

Let C be a *logical clock* i.e. C assigns a *globally unique* time-stamp $C(p)$ to each event p.

Definition (Clock Condition)

Function C satisfies the *clock condition* **if** for any events p, q

$$p \to q \quad \Longrightarrow \quad C(p) < C(q)$$

For a distributed system the *clock condition* holds iff:

1. p_i and p_j are events of P and $p_i \to p_j$ then $C(p_i) < C(p_j)$
2. p is the sending of a message by process P and q is the reception of this message by process Q then $C(p) < C(q)$

Ordering

Let C be a *logical clock* i.e.C assigns a *globally unique* time-stamp $C(p)$ to each event p.

Definition (Clock Condition)

Function C satisfies the *clock condition* **if** for any events p, q

$$p \rightarrow q \quad\Longrightarrow\quad C(p) < C(q)$$

For a distributed system the *clock condition* holds iff:

1. p_i and p_j are events of P and $p_i \rightarrow p_j$ then $C(p_i) < C(p_j)$
2. p is the sending of a message by process P and q is the reception of this message by process Q then $C(p) < C(q)$

$\rightsquigarrow$ a logical clock C that satisfies the clock condition describes a *total order* $a < b$ (with $C(a) < C(b)$) that *embeds* the strict partial order $\rightarrow$

Ordering

Let C be a *logical clock* i.e. C assigns a *globally unique* time-stamp $C(p)$ to each event p.

Definition (Clock Condition)

Function C satisfies the *clock condition* **if** for any events p, q

$$p \rightarrow q \quad \Longrightarrow \quad C(p) < C(q)$$

For a distributed system the *clock condition* holds iff:

1. p_i and p_j are events of P and $p_i \rightarrow p_j$ then $C(p_i) < C(p_j)$
2. p is the sending of a message by process P and q is the reception of this message by process Q then $C(p) < C(q)$

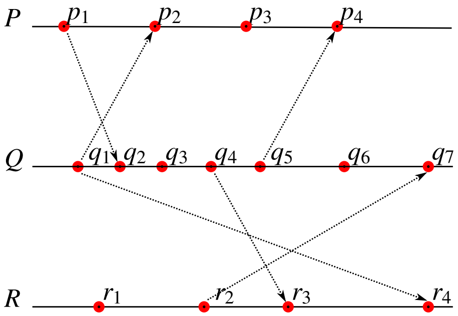
$\leadsto$ a logical clock C that satisfies the clock condition describes a *total order* $a < b$ (with $C(a) < C(b)$) that *embeds* the strict partial order $\rightarrow$

The *set* defined by all C that satisfy the clock condition is exactly the *set* of executions possible in the system.

$\leadsto$ use the process model and $\rightarrow$ to define better consistency model

Defining C Satisfying the Clock Condition

Given:



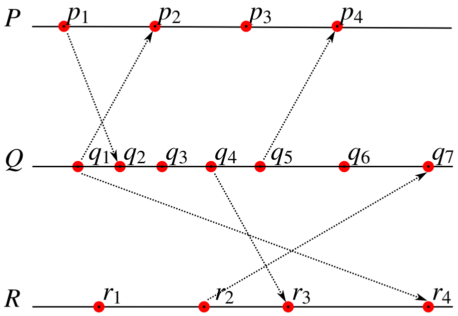
e	p_1	p_2	p_3	p_4
$C(e)$				

e	q_1	q_2	q_3	q_4	q_5	q_6	q_7
$C(e)$							

e	r_1	r_2	r_3	r_4
$C(e)$				

Defining C Satisfying the Clock Condition

Given:



e	p_1	p_2	p_3	p_4
$C(e)$	1	4	7	12

e	q_1	q_2	q_3	q_4	q_5	q_6	q_7
$C(e)$	2	3	5	6	11	13	14

e	r_1	r_2	r_3	r_4
$C(e)$	8	9	10	15

Summing up Happened-Before Relations

We can model concurrency using processes and events:

- there is a *happened-before* relation between the events of each process
- there is a *happened-before* relation between communicating events
- *happened-before* is a strict partial order
- a clock is a total strict order that embeds the *happened-before* partial order

Memory Consistency Models based on the Happened-Before Relation

Happened-Before Based Memory Models

Idea: use happened-before diagrams to model more relaxed memory models.

Given a path through each of the threads of a program:

- consider the actions of each thread as events of a process
- use more processes to model memory
  - here: one process per variable in memory
- $\rightsquigarrow$ concisely represent *some* interleavings

Happened-Before Based Memory Models

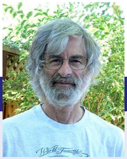
Idea: use happened-before diagrams to model more relaxed memory models.

Given a path through each of the threads of a program:

- consider the actions of each thread as events of a process
- use more processes to model memory
  - here: one process per variable in memory
- $\rightsquigarrow$ concisely represent *some* interleavings

$\rightsquigarrow$ We establish a model for *Sequential Consistency*.

Sequential Consistency



Definition (Sequential Consistency Condition [Lam78])

The result of any execution is the same as if the memory operations

- of each individual processor appear in the order specified by its program
- of all processors joined were executed in some sequential order

Sequential Consistency applied to Multiprocessor Programs:

Given a program with n threads,

1. for fixed event sequences $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $p_0^n, p_1^n, \ldots$ keeping the program order,
2. executions obeying the clock condition on the p_j^i,
3. all executions have the same result

Yet, in other words:

- 1 defines the *execution path* of each thread
- each execution mentioned in 2 is one *interleaving* of processes
- 3 declares that the result of running the threads with these interleavings is always the same.

Working with Sequential Consistency

Sequential Consistency in Multiprocessor Programs:
Given a program with n threads,

1. for fixed event sequences $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $p_0^n, p_1^n, \ldots$ keeping the program order,
2. executions obeying the clock condition on the p_j^i,
3. all executions have the same result

Idea for showing that a system is *not* sequentially consistent:

- pick a result obtained from a program run on a SC system
- pick an execution 1 and a total ordering of all operations 2
- add extra processes to model other system components
- the original order 2 becomes a partial order $\rightarrow$
- show that total orderings C' exist for $\rightarrow$ for which the result differs

Sequential Consistency: Formal Spec [SHW11, p. 25]

Definition (Sequential Consistency)

1. Memory operations in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{Op}_i[a] \leq \mathtt{Op}_i[b]' \Rightarrow \mathtt{Op}_i[a] \sqsubseteq \mathtt{Op}_i[b]'$$

2. A load's value is determined by the latest write wrt. memory order

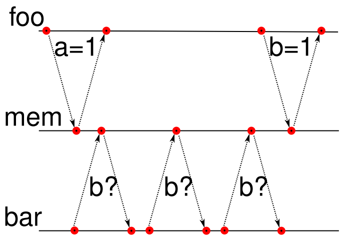
$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq} (\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\})$$

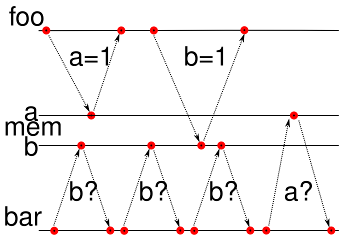
with

- $\mathtt{Op}_i[a]$ any memory access to address a by CPU i
- $\mathtt{Ld}_i[a]$ a load from address a by CPU i
- $\mathtt{St}_i[a]$ a store to address a by CPU i
- Program order $\leq$ being specified by the control flow of the programs executed by their associated CPUs; only orders operations on the same CPU

Weakening the Model

Observation: more concurrency possible, if we model each memory location separately, i.e. as a different process





Sequential consistency still obeyed:

- the accesses of `foo` to `a` occurs before `b`
- the first two read accesses to `b` are in parallel to `a=1`

Conclusion: There is no observable change if accesses to different memory locations can happen in parallel.

Benefits of Sequential Consistency

- concisely represent *all* interleavings that are due to variations in timing
- synchronization using time is uncommon for software

$\rightsquigarrow$ a good model for correct behaviors of concurrent programs

$\rightsquigarrow$ program results besides SC results are undesirable (they contain *races*)

Benefits of Sequential Consistency

- concisely represent *all* interleavings that are due to variations in timing
- synchronization using time is uncommon for software

⇝ a good model for correct behaviors of concurrent programs

⇝ program results besides SC results are undesirable (they contain *races*)

Realistic model for simple hardware architectures:

- sequential consistency model suitable for concurrent processors that acquire *exclusive* access to memory
- processors can speed up computation by using *caches* and still made to maintain sequential consistency

Benefits of Sequential Consistency

- concisely represent *all* interleavings that are due to variations in timing
- synchronization using time is uncommon for software

$\leadsto$ a good model for correct behaviors of concurrent programs

$\leadsto$ program results besides SC results are undesirable (they contain *races*)

Realistic model for simple hardware architectures:

- sequential consistency model suitable for concurrent processors that acquire *exclusive* access to memory
- processors can speed up computation by using *caches* and still made to maintain sequential consistency

Not realistic for elaborate hardware with out-of-order stores:

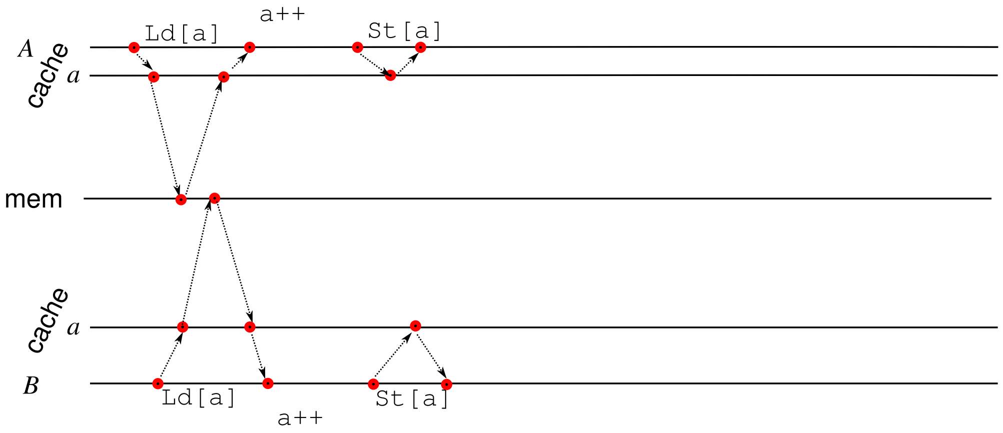
- what other processors see is determined by complex optimizations to cacheline management

$\leadsto$ internal workings of caches

Introducing Caches: The MESI Protocol

Introducing Caches

Idea: each cache line one process



Observations:

⚠ naive replication of memory in cache lines creates *incoherency*

Cache Coherency: Formal Spec [SHW11, p. 14]

Definition (Cache Coherency)

1. Memory operations in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{Op}_i[a] \leq \mathtt{Op}_i[a]' \Rightarrow \mathtt{Op}_i[a] \sqsubseteq \mathtt{Op}_i[a]'$$

2. A load's value is determined by the latest write wrt. memory order

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

- This definition superficially looks close to the definition of SC – except that it covers only singular memory locations instead of all memory locations accessed in a program
- Caches and memory can communicate using messaging, following some particular protocol to establish cache coherency
  ($\rightsquigarrow$ *Cache Coherence Protocol*)

The MESI Cache Coherence Protocol: States [PP84]

Processors use caches to avoid a costly round-trip to RAM for every memory access.

- programs often access the same memory area repeatedly (e.g. stack)
- keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

$$\begin{tikzcd}
M \arrow[r, shift left] \arrow[d, shift left] \arrow[dr, shift left] & E \arrow[l, shift left] \arrow[d, shift left] \arrow[dl, shift left] \\
S \arrow[u, shift left] \arrow[r, shift left] \arrow[ur, shift left] & I \arrow[u, shift left] \arrow[l, shift left] \arrow[ul, shift left]
\end{tikzcd}$$

Each cache line is in one of the states M, E, S, I:

The MESI Cache Coherence Protocol: States [PP84]

Processors use caches to avoid a costly round-trip to RAM for every memory access.

- programs often access the same memory area repeatedly (e.g. stack)
- keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

$$\begin{tikzcd}
M \arrow[r, shift left] \arrow[d, shift left] \arrow[dr, shift left] & E \arrow[l, shift left] \arrow[d, shift left] \arrow[dl, shift left] \\
S \arrow[u, shift left] \arrow[r, shift left] \arrow[ur, shift left] & \boxed{I} \arrow[l, shift left] \arrow[u, shift left] \arrow[ul, shift left]
\end{tikzcd}$$

Each cache line is in one of the states M, E, S, I:

I**:** it is *invalid* and is ready for re-use

The MESI Cache Coherence Protocol: States [PP84]

Processors use caches to avoid a costly round-trip to RAM for every memory access.

- programs often access the same memory area repeatedly (e.g. stack)
- keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

$$\begin{tikzcd}
M \arrow[r, shift left] \arrow[d, shift left] \arrow[dr, shift left] & E \arrow[l, shift left] \arrow[d, shift left] \arrow[dl, shift left] \\
\boxed{S} \arrow[u, shift left] \arrow[r, shift left] \arrow[ur, shift left] & I \arrow[l, shift left] \arrow[u, shift left] \arrow[ul, shift left]
\end{tikzcd}$$

Each cache line is in one of the states M, E, S, I:

I**:** it is *invalid* and is ready for re-use

S**:** other caches have an identical copy of this cache line, it is *shared*

The MESI Cache Coherence Protocol: States [PP84]

Processors use caches to avoid a costly round-trip to RAM for every memory access.

- programs often access the same memory area repeatedly (e.g. stack)
- keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

$$\begin{tikzcd}
M \arrow[r, shift left] \arrow[d, shift left] \arrow[dr, shift left] & \boxed{E} \arrow[l, shift left] \arrow[d, shift left] \arrow[dl, shift left] \\
S \arrow[u, shift left] \arrow[r, shift left] \arrow[ur, shift left] & I \arrow[l, shift left] \arrow[u, shift left] \arrow[ul, shift left]
\end{tikzcd}$$

Each cache line is in one of the states M, E, S, I:

I**:** it is *invalid* and is ready for re-use

S**:** other caches have an identical copy of this cache line, it is *shared*

E**:** the content is in no other cache; it is *exclusive* to this cache and can be overwritten without consulting other caches

The MESI Cache Coherence Protocol: States [PP84]

Processors use caches to avoid a costly round-trip to RAM for every memory access.

- programs often access the same memory area repeatedly (e.g. stack)
- keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

$$\begin{tikzcd}
\boxed{M} \arrow[r, shift left] \arrow[d, shift left] \arrow[dr, shift left] & E \arrow[l, shift left] \arrow[d, shift left] \arrow[dl, shift left] \\
S \arrow[u, shift left] \arrow[r, shift left] \arrow[ur, shift left] & I \arrow[l, shift left] \arrow[u, shift left] \arrow[ul, shift left]
\end{tikzcd}$$

Each cache line is in one of the states M, E, S, I:

I**:** it is *invalid* and is ready for re-use

S**:** other caches have an identical copy of this cache line, it is *shared*

E**:** the content is in no other cache; it is *exclusive* to this cache and can be overwritten without consulting other caches

M**:** the content is exclusive to this cache and has furthermore been *modified*

The MESI Cache Coherence Protocol: States [PP84]

Processors use caches to avoid a costly round-trip to RAM for every memory access.

- programs often access the same memory area repeatedly (e.g. stack)
- keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

$$\begin{tikzcd}
M \arrow[r, shift left] \arrow[d, shift left] \arrow[dr, shift left] & E \arrow[l, shift left] \arrow[d, shift left] \arrow[dl, shift left] \\
S \arrow[r, shift left] \arrow[u, shift left] \arrow[ur, shift left] & I \arrow[l, shift left] \arrow[u, shift left] \arrow[ul, shift left]
\end{tikzcd}$$

Each cache line is in one of the states M, E, S, I:

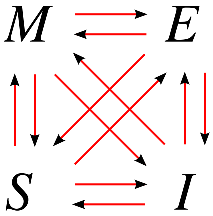
- I**:** it is *invalid* and is ready for re-use
- S**:** other caches have an identical copy of this cache line, it is *shared*
- E**:** the content is in no other cache; it is *exclusive* to this cache and can be overwritten without consulting other caches
- M**:** the content is exclusive to this cache and has furthermore been *modified*

$\leadsto$ the global state of cache lines is kept consistent by sending *messages*

The MESI Cache Coherence Protocol: Messages

Moving data between caches is coordinated by sending messages [McK10]:

- *Read:* sent if CPU needs to read from an address
- *Read Response:* when in state E or S, response to a *Read* message, carries the data for the requested address
- *Invalidate:* asks others to evict a cache line
- *Invalidate Acknowledge:* reply indicating that a cache line has been evicted
- *Read Invalidate:* like *Read* + *Invalidate* (also called “read with intend to modify”)
- *Writeback: Read Response* when in state M, as a side effect noticing main memory about modifications to the cacheline, changing sender’s state to S



We mostly consider messages between processors. Upon *Read Invalidate*, a processor replies with *Read Response*/*Writeback* before the *Invalidate Acknowledge* is sent.

MESI Example

Consider how the following code might execute:

Thread A

```
a = 1;       // A.1
b = 1;       // A.2
```

Thread B

```
while (b == 0) {};   // B.1
assert(a == 1);      // B.2
```

- in all examples, the initial values of variables are assumed to be 0
- suppose that `a` and `b` reside in different cache lines
- assume that a cache line is larger than the variable itself
- we write the content of a cache line as
  - Mx: modified, with value x
  - Ex: exclusive, with value x
  - Sx: shared, with value x
  - I: invalid

MESI Example (I)

Thread A

```
a = 1;      // A.1
b = 1;      // A.2
```

Thread B

```
while (b == 0) {};   // B.1
assert(a == 1);      // B.2
```

statement	CPU A		CPU B		RAM		message
	a	b	a	b	a	b	
A.1	I	I	I	I	0	0	read invalidate of a from CPU A
	I	I	I	I	0	0	invalidate ack. of a from CPU B
	I	I	I	I	0	0	read response of a=0 from RAM
B.1	M 1	I	I	I	0	0	read of b from CPU B
	M 1	I	I	I	0	0	read response with b=0 from RAM
B.1	M 1	I	I	E 0	0	0	
A.2	M 1	I	I	E 0	0	0	read invalidate of b from CPU A
	M 1	I	I	E 0	0	0	read response of b=0 from CPU B
	M 1	S 0	I	S 0	0	0	invalidate ack. of b from CPU B
	M 1	M 1	I	I	0	0	

MESI Example (II)

Thread A

```
a = 1;      // A.1
b = 1;      // A.2
```

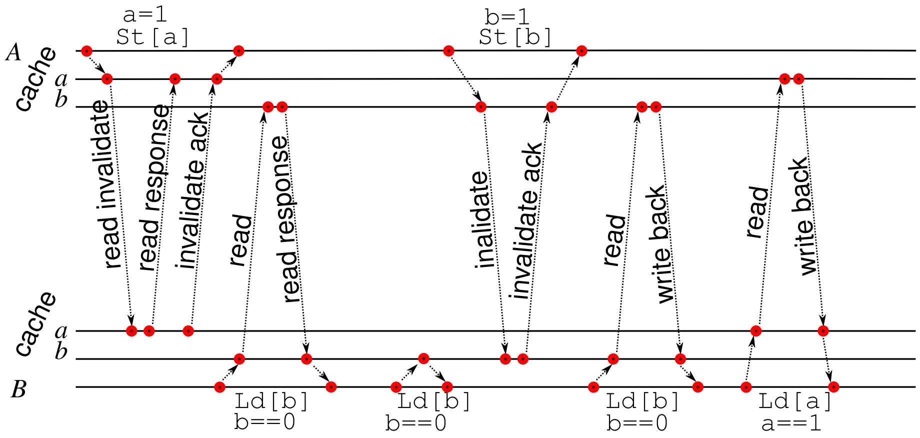
Thread B

```
while (b == 0) {};   // B.1
assert(a == 1);      // B.2
```

statement	CPU A		CPU B		RAM		message
	a	b	a	b	a	b	
B.1	M 1	M 1	I	I	0	0	read of b from CPU B
	M 1	M 1	I	I	0	0	write back of b=1 from CPU A
B.2	M 1	S 1	I	S 1	0	1	read of a from CPU B
	M 1	S 1	I	S 1	0	1	write back of a=1 from CPU A
	S 1	S 1	S 1	S 1	1	1	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
A.1	S 1	S 1	S 1	S 1	1	1	invalidate of a from CPU A
	S 1	S 1	I	S 1	1	1	invalidate ack. of a from CPU B
	M 1	S 1	I	S 1	1	1	

MESI Example: Happened Before Model

Idea: each cache line one process, A caches b=0 as E, B caches a=0 as E

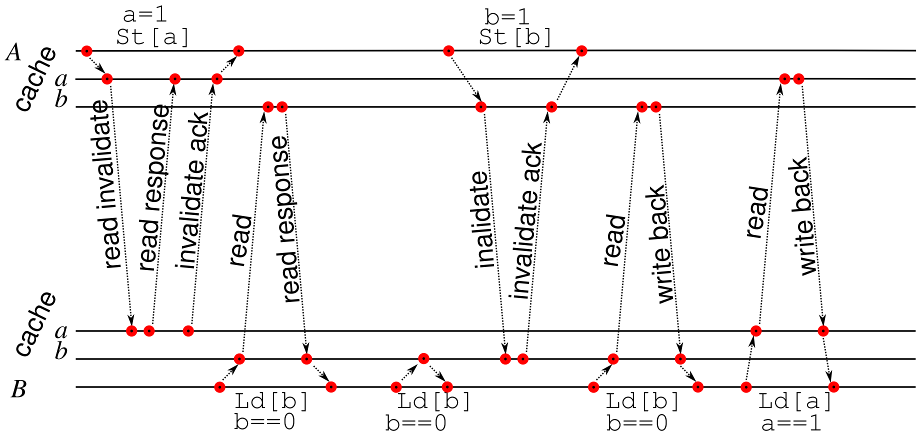


Observations:

- each memory access must complete before executing next instruction ⇝ add edge

MESI Example: Happened Before Model

Idea: each cache line one process, A caches b=0 as E, B caches a=0 as E



Observations:

- each memory access must complete before executing next instruction $\leadsto$ add edge
- second execution of test `b==0` stays within cache $\leadsto$ no traffic

Summary: MESI Cache Coherence Protocol

Sequential Consistency:

- specifies that the system must appear to execute all threads' loads and stores to *all memory locations* in a total order that respects the program order of each thread
- a characterization of well-behaved programs
- a model for differing speed of execution
- for fixed paths through the threads *and* a total order between accesses to the same variables: executions can be illustrated by a happened-before diagram with one process per variable

Cache Coherency:

- A *cache coherent* system must appear to execute all threads' loads and stores to a *single memory location* in a total order that respects the program order of each thread
- MESI cache coherence protocol ensures SC for processors with caches

Introducing Store Buffers: Out-Of-Order Stores

Out-of-Order Execution

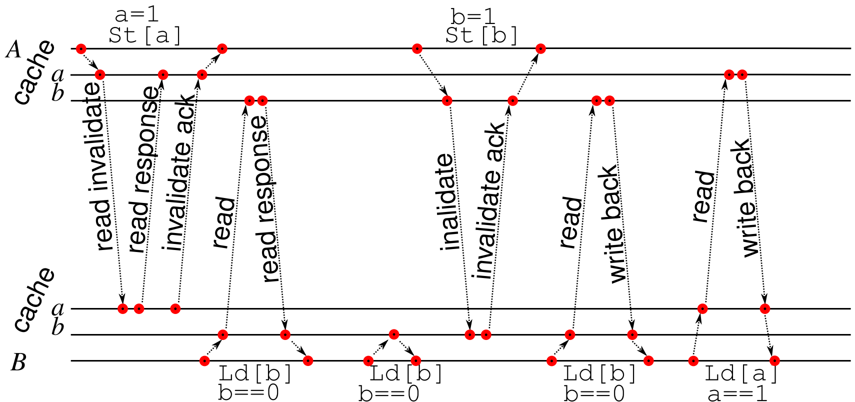
⚠ performance problem: writes always stall

Thread A

```
a = 1;       // A.1
b = 1;       // A.2
```

Thread B

```
while (b == 0) {};   // B.1
assert(a == 1);      // B.2
```



Out-of-Order Execution

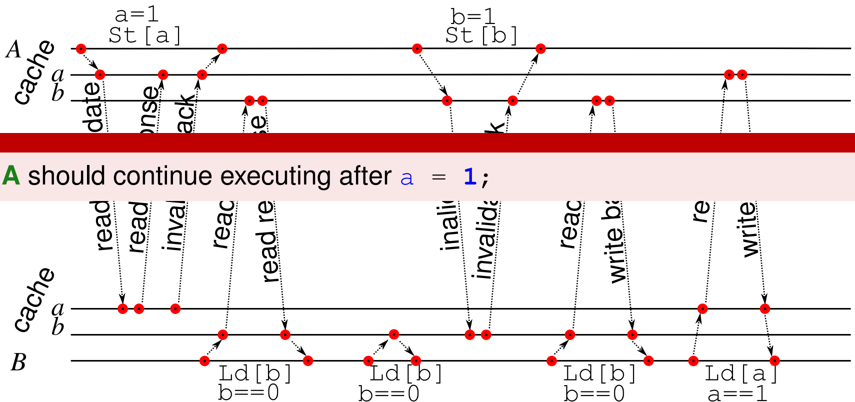
⚠ performance problem: writes always stall

Thread A

```
a = 1;        // A.1
b = 1;        // A.2
```

Thread B

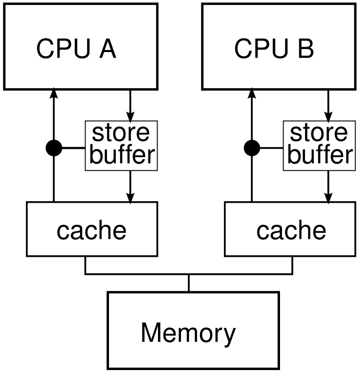
```
while (b == 0) {};   // B.1
assert(a == 1);      // B.2
```



⇝ CPU **A** should continue executing after `a = 1;`

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

TSO Model: Formal Spec [SI92] [SHW11, p. 42]

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 ($\leq$) are embedded into the memory order ($\sqsubseteq$)

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

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

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

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

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

Particularly, one ordering property from SC is not guaranteed:

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

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

Happened-Before Model for TSO

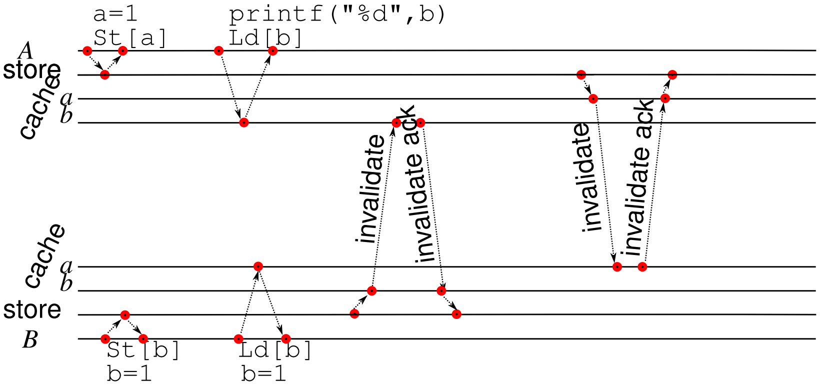
Thread A

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

Thread 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 CPU, 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:

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

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

$\leadsto$ use fences only when necessary

PSO Model: Formal Spec [SI92] [SHW11, p. 58]

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 ($\sqsubseteq$)

$$\texttt{St}_i[a] \leq \texttt{sfence()} \leq \texttt{St}_i[b] \Rightarrow \texttt{St}_i[a] \sqsubseteq \texttt{St}_i[b]$$

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 another 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:

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

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

Happened-Before Model for PSO

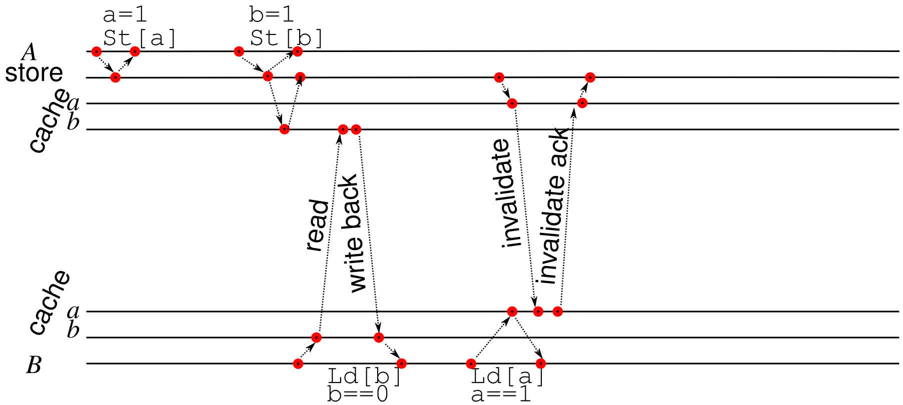
Thread A

```
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

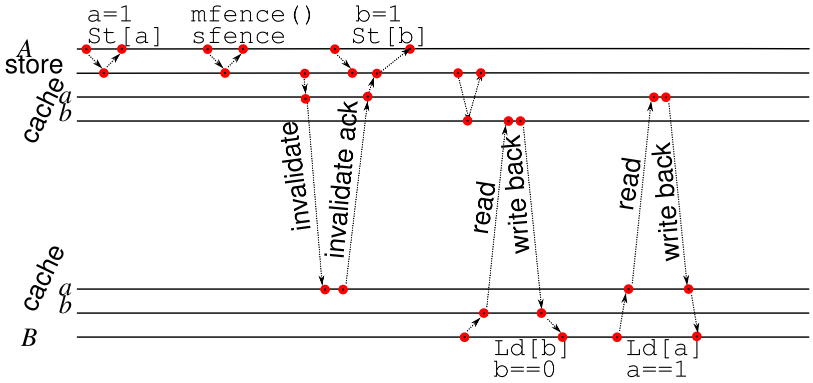
Thread A

```
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

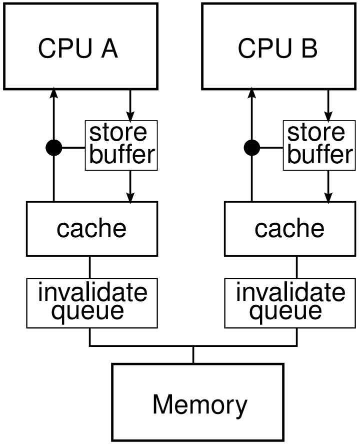
Relaxed Memory Order

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, p. 290]

Definition (Relaxed Memory Order)

1. Fenced memory accesses in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)

$$\mathtt{Op}_i[a] \leq \mathtt{mfence()} \leq \mathtt{Op}_i[b] \Rightarrow \mathtt{Op}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$

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

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

3. Operations dependent on a load (wrt. *dependence* $\rightarrow$) are embedded in the memory order ($\sqsubseteq$)

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

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

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

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

- Memory access to the same address: $\mathtt{St}_i[a] \leq \mathtt{Ld}_i[a] \quad \Rightarrow \quad \mathtt{St}_i[a] \rightarrow \mathtt{Ld}_i[a]$
- Register reads are dependent on latest register writes:

$$\mathtt{Ld}_i[a]'' \underset{\leq}{=max} (\mathtt{Ld}_i[a]' \mid targetreg(\mathtt{Ld}_i[a]') = srcreg(\mathtt{St}_i[b]) \wedge \mathtt{Ld}_i[a]' \leq \mathtt{St}_i[b]) \quad \Rightarrow \quad \mathtt{Ld}_i[a]'' \rightarrow \mathtt{St}_i[b]$$

- Stores within branched blocks are dependent on branch conditionals:

$$(\mathtt{Op}_i[a] \leq \mathtt{St}_i[b]) \wedge \mathtt{Op}_i[a] \rightarrow condbranch \leq \mathtt{St}_i[b] \quad \Rightarrow \quad \mathtt{Op}_i[a] \rightarrow \mathtt{St}_i[b]$$

Happened-Before Model for Invalidate Queues

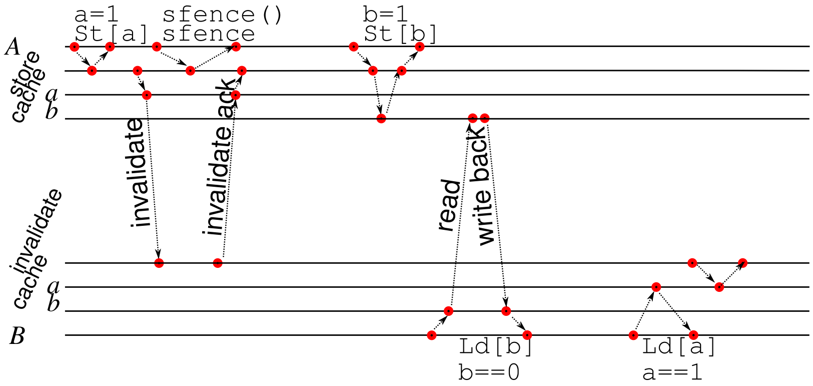
Thread A

```
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)

$\rightsquigarrow$ match each write barrier in one process with a read barrier in another process

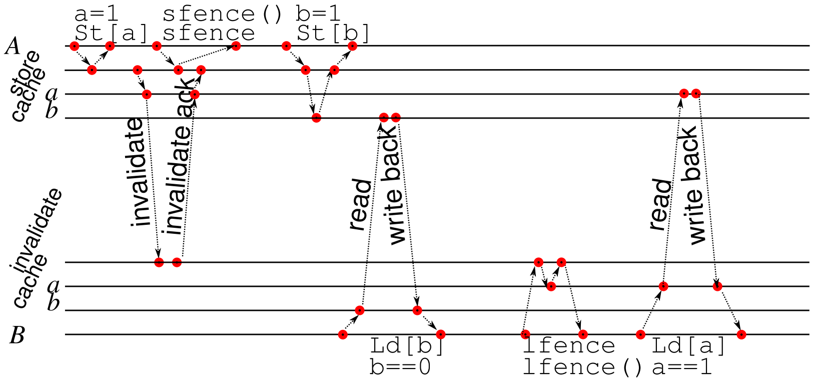
Happened-Before Model for Read Barriers

Thread A

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

Thread B

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



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

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

$\leadsto$ `flag[i]` is a *lock* and may be implemented as such

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

$\rightsquigarrow$ `flag[i]` is a *lock* and may be implemented as such

- if P_{1-i} also wants to enter, wait for `turn` to be set to `i`

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

$\rightsquigarrow$ `flag[i]` is a *lock* and may be implemented as such

- if P_{1-i} also wants to enter, wait for `turn` to be set to `i`
- while waiting for `turn`, reset `flag[i]` to enable P_{1-i} to progress

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

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
- insert a write memory barrier `sfence()` after writing a variable that is read in the other thread

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
- insert a write memory barrier `sfence()` after writing a variable that is read in the other thread
- the `lfence()` of the first iteration of each loop may be combined with the preceding `sfence()` to an `mfence()`

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

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

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

Discussion

Memory barriers reside at the lowest level of synchronization primitives.

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

$\leadsto$ protocol implementations

$\leadsto$ 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 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.

⚠ having another thread executing `x = 0;` changes observable behaviour depending on optimizing or not

⇝ Compiler also depends on consistency guarantees
⇝ Demand for Memory Models on language level

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

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
- *Java*-Compilers even generate barriers around accesses to `volatile` variables

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

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

$\leadsto$ use a bus locally, use point-to-point links globally: *NUMA*

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:

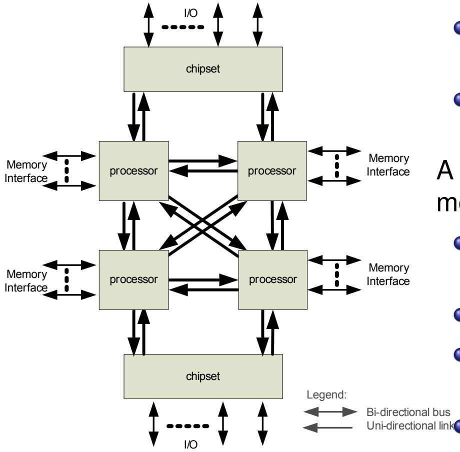
- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus

⇝ use a bus locally, use point-to-point links globally: *NUMA*

- *non-uniform memory access* partitions the memory amongst CPUs
- a directory states which CPU holds a memory region
- Interprocess communication between Cache-Controllers (*ccNUMA*): onchip on Opteron or in chipset on Itanium

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.

Daniel J. Sorin, Mark D. Hill, and David A. Wood.
A Primer on Memory Consistency and Cache Coherence.
Morgan & Claypool Publishers, 1st edition, 2011.

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.

Cache Coherence vs. Memory Consistency Models

- *Sequential Consistency* specifies that the system must appear to execute all threads' loads and stores to *all memory locations* in a total order that respects the program order of each thread
- A *cache coherent* system must appear to execute all threads' loads and stores to a *single memory location* in a total order that respects the program order of each thread

All discussed memory models (SC, TSO, PSO, RMO) provide cache coherence!