Script generated by TTT

Title: Petter: Programmiersprachen (21.10.2015)

Date: Wed Oct 21 14:20:20 CEST 2015

Duration: 91:39 min

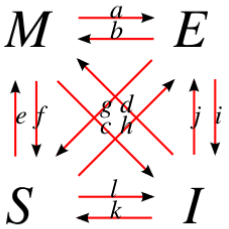
Pages: 35

Introducing Caches: The MESI Protocol

The MESI Protocol: States

Processors (and also: GPUs, intelligent I/O devices) 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

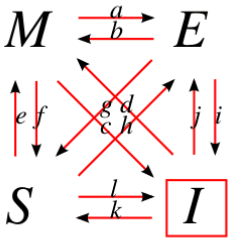


Each cache line is in one of the states M, E, S, I:

The MESI Protocol: States

Processors (and also: GPUs, intelligent I/O devices) 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



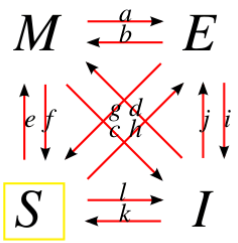
Each cache line is in one of the states M, E, S, I:

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

The MESI Protocol: States

Processors (and also: GPUs, intelligent I/O devices) 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



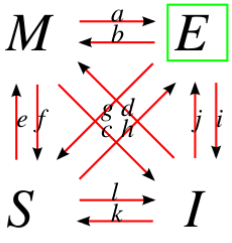
Each cache line is in one of the states M, E, S, I:

- 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 Protocol: States

Processors (and also: GPUs, intelligent I/O devices) 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



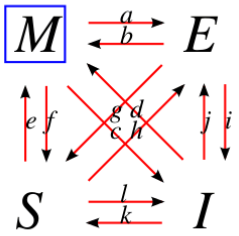
Each cache line is in one of the states M, E, S, I:

- 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 Protocol: States

Processors (and also: GPUs, intelligent I/O devices) 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



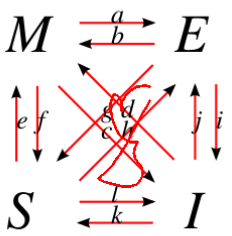
Each cache line is in one of the states M, E, S, I:

- 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 Protocol: States

Processors (and also: GPUs, intelligent I/O devices) 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



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

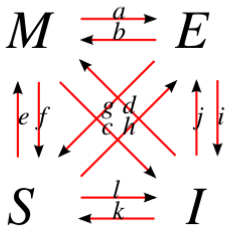
- I: it is *invalid* and is ready for re-use
- S: other caches have an identical copy of this cache line, it is *shared*
- E: the content is in no other cache; it is *exclusive* to this cache and can be overwritten without consulting other caches
- M: the content is exclusive to this cache and has furthermore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

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

The MESI Protocol: Messages

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

- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address
- *Invalidate:* asks others to evict a cache line
- *Invalidate Acknowledge:* reply indicating that an address has been evicted
- *Read Invalidate:* like *Read* + *Invalidate* (also called "read with intend to modify")
- *Writeback:* info on what data has been sent to main memory



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
```

state-ment	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
A.1	I	I	I	I	0	0	
	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
```

state-ment	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
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 (II)

Thread A

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

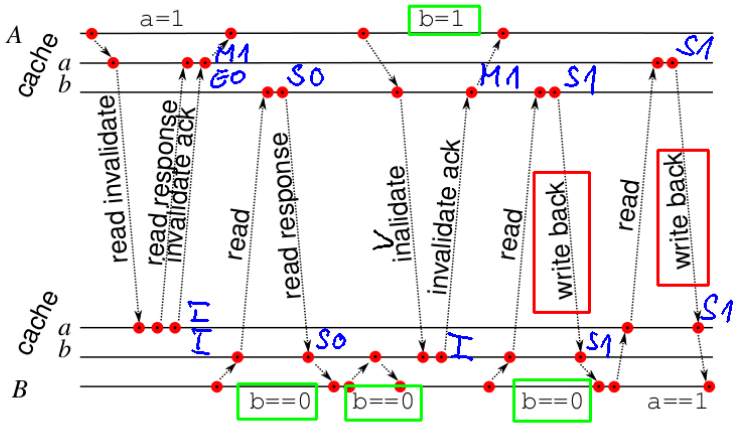
Thread B

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

state-ment	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
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:

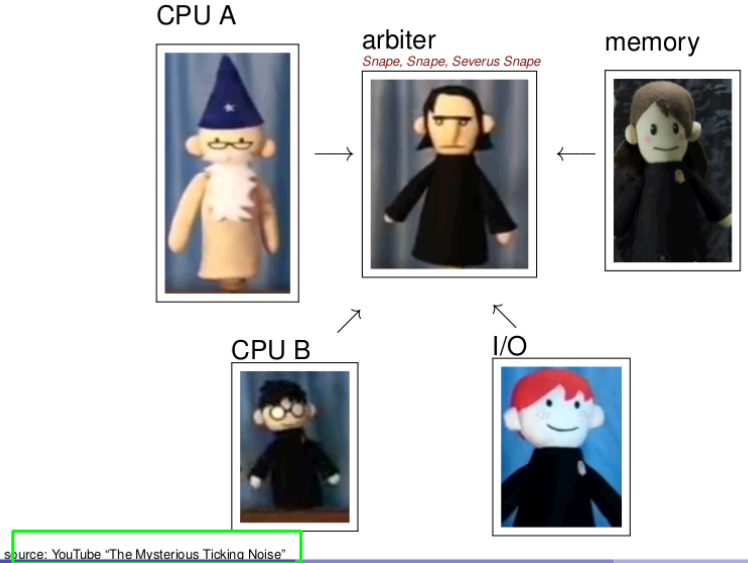
- each memory access must complete before executing next instruction
  ⇝ add edge

Can MESI Messages Collide?

If two processors emit a message at the same time, the protocol might break.
Access to common bus is coordinated by an *arbiter*.

Can MESI Messages Collide?

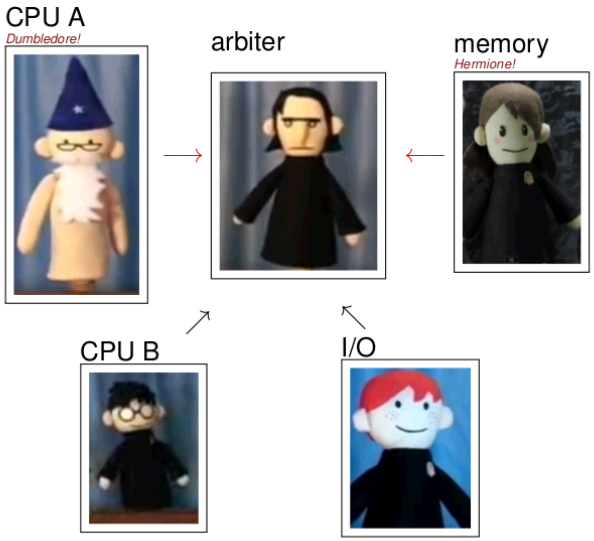
If two processors emit a message at the same time, the protocol might break.
Access to common bus is coordinated by an *arbiter*.



source: YouTube "The Mysterious Ticking Noise"

Can MESI Messages Collide?

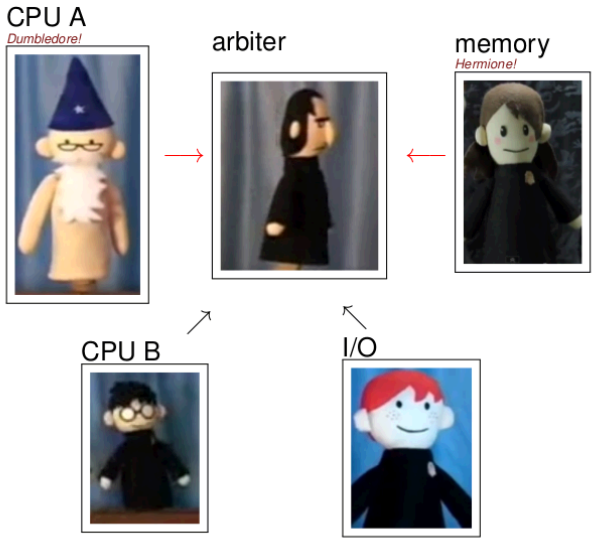
If two processors emit a message at the same time, the protocol might break.
Access to common bus is coordinated by an *arbiter*.



source: YouTube "The Mysterious Ticking Noise"

Can MESI Messages Collide?

If two processors emit a message at the same time, the protocol might break.
Access to common bus is coordinated by an *arbiter*.

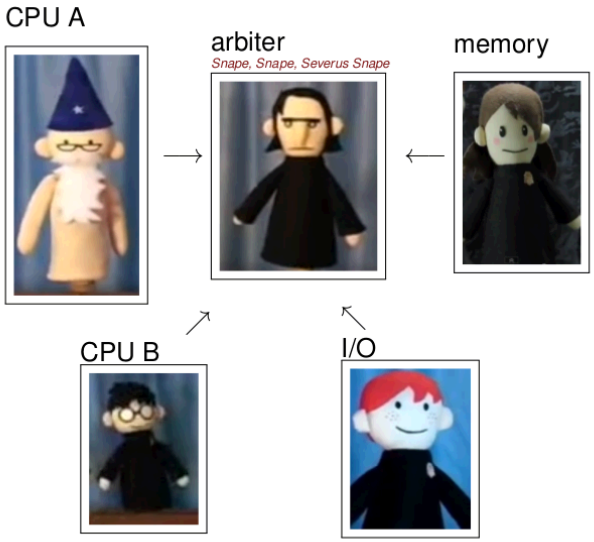


source: YouTube "The Mysterious Ticking Noise"

Memory Consistency The MESI Protocol 29 / 54

Can MESI Messages Collide?

If two processors emit a message at the same time, the protocol might break.
Access to common bus is coordinated by an *arbiter*.



source: YouTube "The Mysterious Ticking Noise"

Memory Consistency The MESI Protocol 29 / 54

Summary: MESI cc-Protocol

Sequential consistency:

- a characterization of well-behaved programs
- a model for different speed of execution
- for fixed paths through the threads *and* a total order between accesses to the same variable: executions can be illustrated by happened-before diagram with one process per variable
- MESI cache coherence protocol ensures SC for processors with caches

Introducing Store Buffers: Out-Of-Order-Writes

Out-of-Order Execution

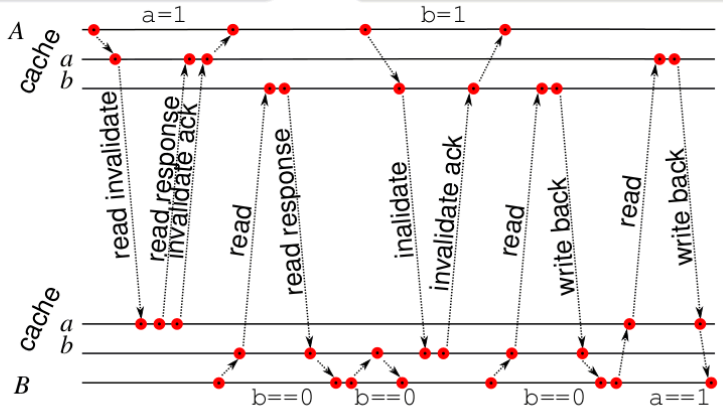
⚠ performance problem: writes always stall

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

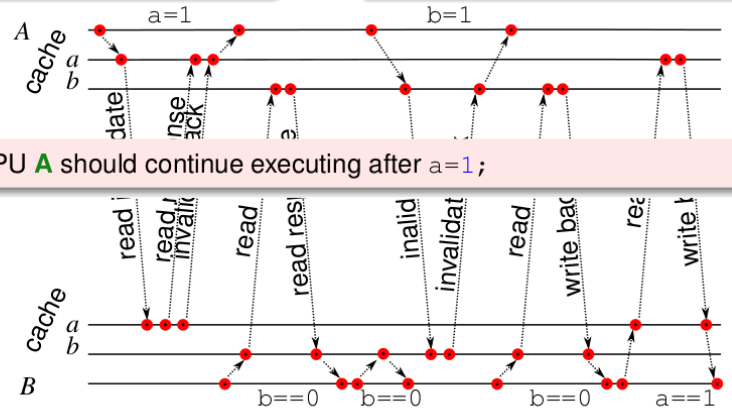
⚠ performance problem: writes always stall

Thread A

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

Thread B

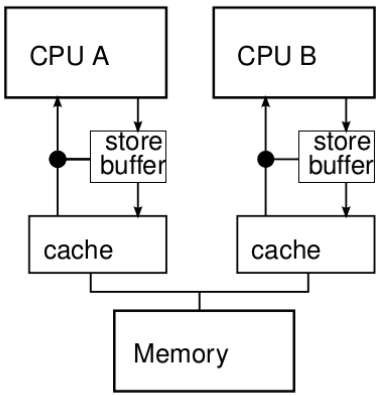
```
while (b == 0) {};   // B.1
assert(a == 1);      // B.2
```



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

Store Buffers

Goal: continue execution after *cache-miss* write operation



- put each write into a *store buffer* and trigger fetching of cache line
- once a cache line has arrived, apply relevant writes
  - today, a store buffer is always a *queue* [OSS09]
  - two writes to the same location are not merged
- ⚠ sequential consistency per CPU is violated unless
  - each read checks store buffer before cache
  - on hit, return the youngest value that is waiting to be written

What about sequential consistency for the whole system?

Happened-Before Model for Store Buffers

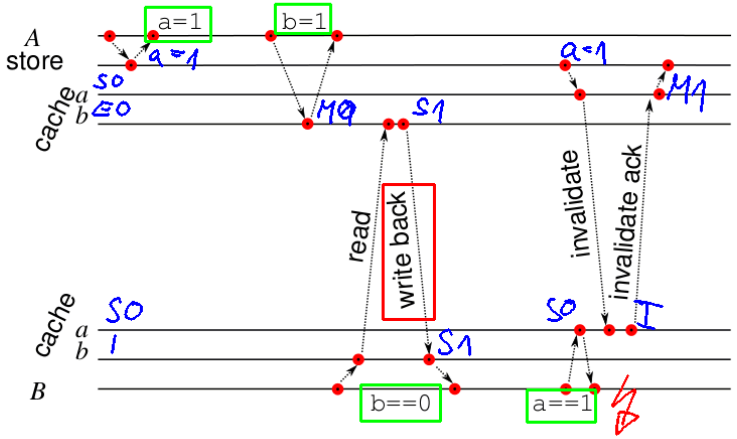
Thread A

```
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 *is desirable* and should not be prohibited in general.

- store buffers render programs incorrect that assume sequential consistency between *different* CPUs
- whenever two stores in one CPU must appear *in sequence at a different CPU*, an explicit *write barrier* has to be inserted
- x86 CPUs provide the sfence instruction
- 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
- a write barrier after each write gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)

⇝ use (write) barriers only when necessary

Happened-Before Model for Write Fences

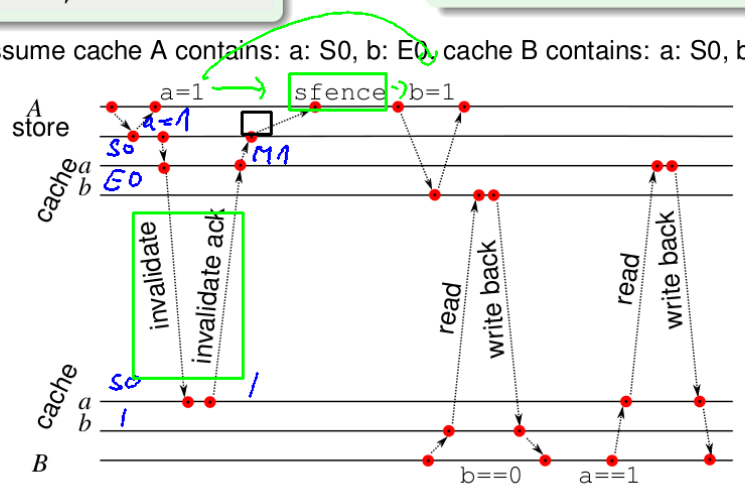
Thread A

```
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



Introducing Invalidate Queues: O-O-O Reads

Happened-Before Model for Write Fences

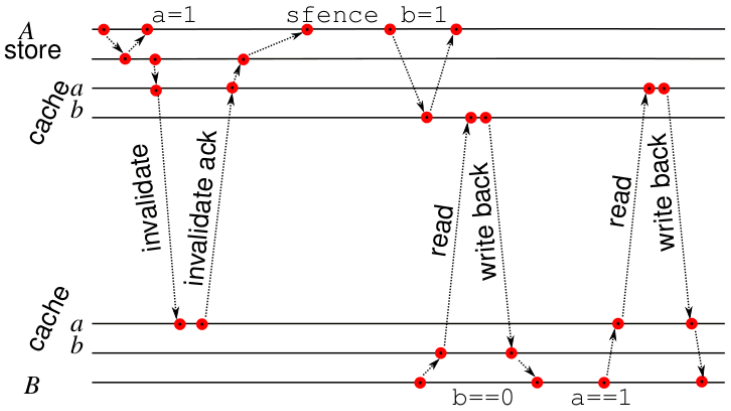
Thread A

```
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

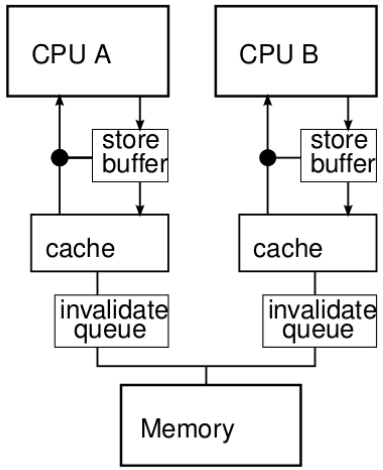


Introducing Invalidate Queues: O-O-O Reads

Invalidate Queue

Invalidation of cache lines is costly:

- all CPUs in the system need to send an acknowledge
- invalidating a cache line competes with CPU accesses
- a cache-intense computation can fill up store buffers in other CPUs



⇝ immediately acknowledge an invalidation and apply them 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 read and writes do *not* consult the invalidate queue
- What about sequential consistency?

Happened-Before Model for Invalidate Queues

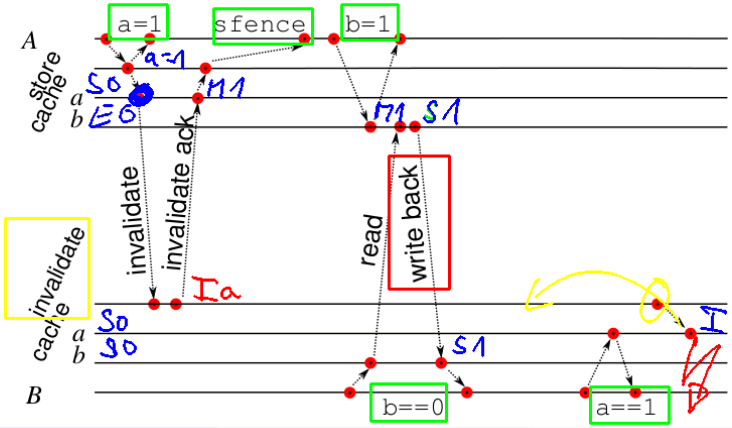
Thread A

```
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
- Intel x86 CPUs provide the `lfence` instruction
- 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

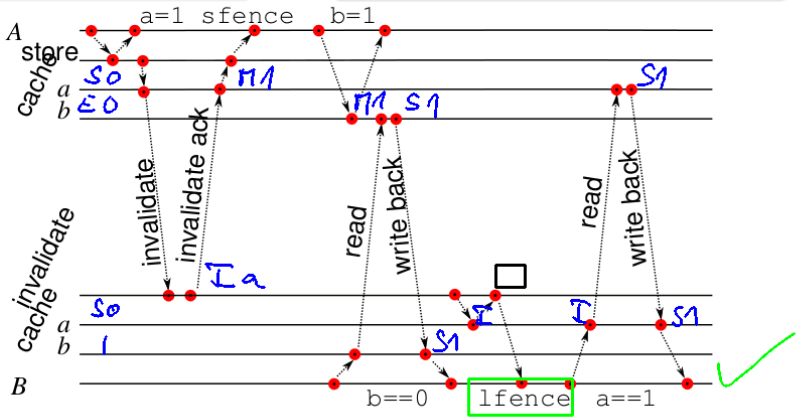
Happened-Before Model for Read Barriers

Thread A

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

Thread B

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



Summary: Weakly-Ordered Memory Models

Modern CPUs use a *weakly-ordered memory model*:

- reads and writes are not synchronized unless requested by the user
- many kinds of memory barriers exist with subtle differences
- most systems provide a barrier that is both, read and write (e.g `mfence` on x86)
- ahead-of-time imperative languages can use memory barriers, but compiler optimizations may render programs incorrect
- use the `volatile` keyword in C/C++
- in the latest C++ standard, an access to a `volatile` variable will automatically insert a memory barrier
- otherwise, inline assembler has to be used

⇝ memory barriers are the “lowest-level” of synchronization

Example: The Dekker Algorithm on SMP Systems