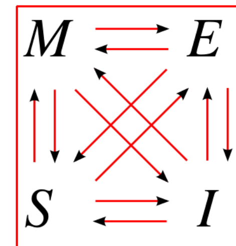
Script generated by TTT

Title:	Petter: Programmiersprachenh (24.10.2018)
Date:	Wed Oct 24 14:09:04 CEST 2018
Duration:	94:44 min
Pages:	16

The MESI Protocol: States [?]

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

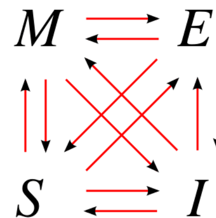


Each cache line is in one of the states M, E, S, I:

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:* 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
```

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	*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
```

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

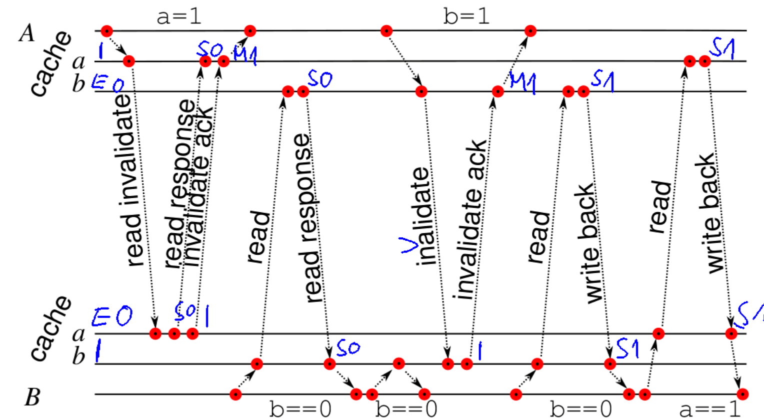
```
while (b == 0) {};    // B.1
assert(a == 1);       // B.2
```

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	

Memory Consistency The MESI Protocol 24 / 52

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

Summary: MESI cache coherence protocol

Sequential consistency:

- 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 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 Stores

Out-of-Order Execution

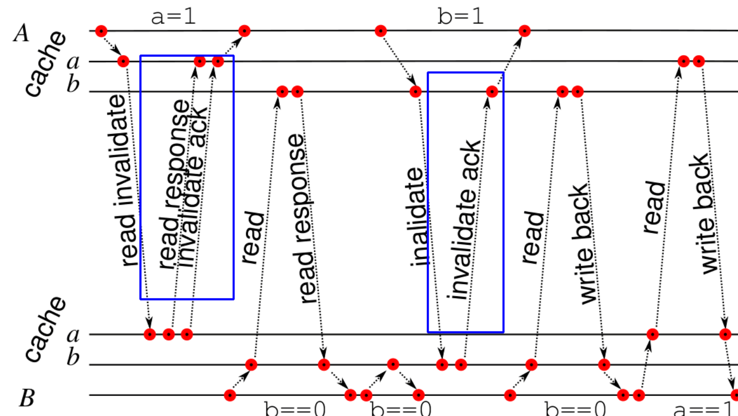
⚠ performance problem: writes always stall

Thread A

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

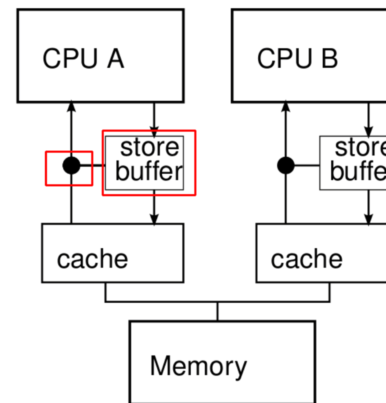
Thread B

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



Store Buffers and *T*otal *S*tore *O*rdering [?]

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

  ⇝ *TSO*

Excursion : non-FIFO store buffers (⇝ *Sparc/PSO*)

TSO Model: Formal Spec [?]

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 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!

⇝ What about sequential consistency for the whole system?

Happened-Before Model for Store Buffers

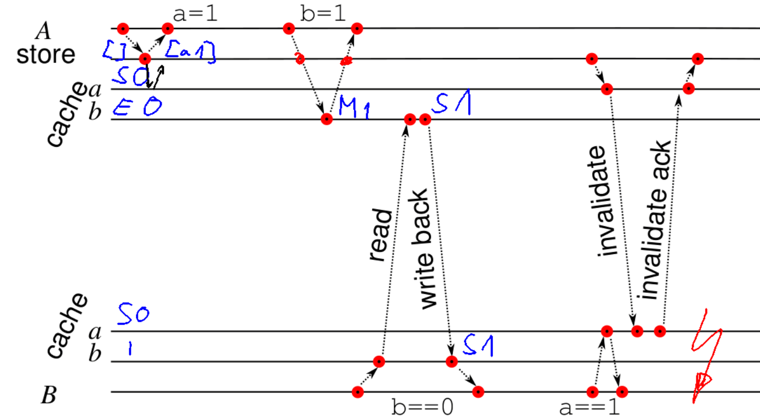
Thread A

```
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
  - 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
- x86 CPUs provide the `sfence` instruction
- 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 Barriers

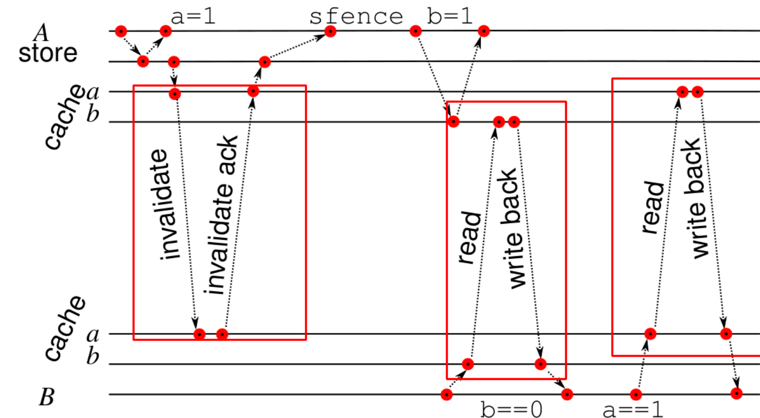
Thread A

```
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



Introducing Invalidate Queues: O-o-O Reads