Script generated by TTT

Title: Petter: Programmiersprachen (08.10.2014)

Date: Wed Oct 08 14:24:25 CEST 2014

Duration: 79:53 min

Pages: 82

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Memory Consistency

Dr. Axel Simon and Dr. Michael Petter
Winter term 2014

Need for Concurrency

Consider two processors:

- in 1997 the *Pentium P55C* had 4.5M transistors
- in 2006 the *Itanium 2* had 1700M transistors

⇝ Intel could have built a processor with 256 Pentium cores in 2006

Need for Concurrency

Consider two processors:

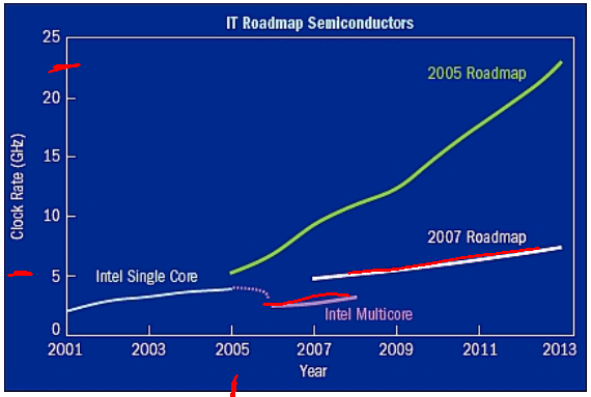
- in 1997 the *Pentium P55C* had 4.5M transistors
- in 2006 the *Itanium 2* had 1700M transistors

⇝ Intel could have built a processor with 256 Pentium cores in 2006
However:

- most programs are not inherently parallel
  - ⇝ parallelizing a program is between difficult and impossible
- correctly communicating between different cores is challenging
  - ⇝ correctness of concurrent communication is very hard
  - low-level aspects: locking algorithms must be correct
  - high-level aspects: program may deadlock
- a program on n cores runs $m \ll n$ times faster
  - ⇝ all effort is voided if program runs no faster
  - distributing work load is application specific

The free lunch is over

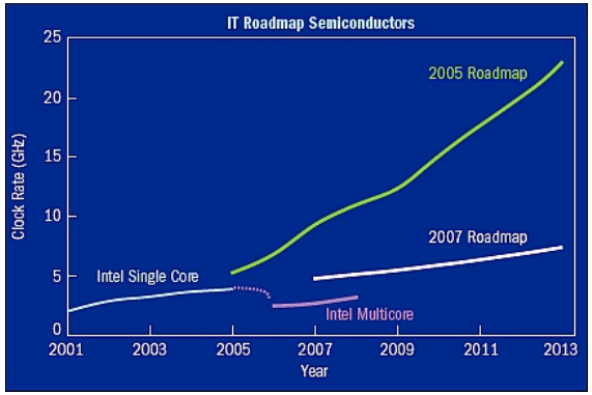
Single processors cannot be made much faster due to physical limitations.



Source: D. Patterson, UC–Berkeley

The free lunch is over

Single processors cannot be made much faster due to physical limitations.



Source: D. Patterson, UC–Berkeley

But Moore's law still holds for the number of transistors:

- they double every 18 months for the foreseeable future
- may translate into doubling the number of cores
- programs have to become parallel

Concurrency for the Programmer

How is concurrency exposed in a programming language?

1. spawning of new concurrent computations
2. communication between threads

Concurrency for the Programmer

How is concurrency exposed in a programming language?

1. spawning of new concurrent computations
2. communication between threads

Communication can happen in many ways:

- communication via shared memory *(this lecture)*
- atomic transactions on shared memory
- message passing

Learning Outcomes

1. Happened-before Partial Order
2. Sequential Consistency
3. The MESI Cache Model
4. Weak Consistency
5. Memory Barriers

Communication between Cores

We consider the concurrent execution of these functions:

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

Communication between Cores

We consider the concurrent execution of these functions:

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`

Communication between Cores

We consider the concurrent execution of these functions:

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 one before executing the `assert` statement

Communication between Cores

We consider the concurrent execution of these functions:

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 one before executing the `assert` statement
- the `assert` statement should always hold
- here the code is *correct* if the `assert` holds

Communication between Cores

We consider the concurrent execution of these functions:

Thread A

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

Thread B

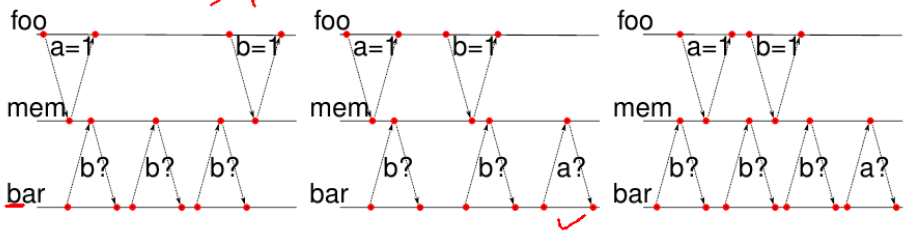
```
void bar(void) {
  while (b == 0) {};
  assert(a == 1);
}
```

- initial state of `a` and `b` is 0
- A writes `a` before it writes `b`
- B should see `b` go to one before executing the `assert` statement
- the `assert` statement should always hold
- here the code is *correct* if the `assert` holds

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

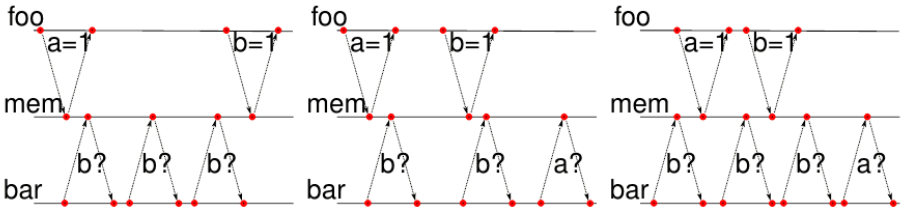
Strict Consistency

Assuming `foo` and `bar` are started on two cores operating in lock-step.
Then *one* of the following may happen:



Strict Consistency

Assuming `foo` and `bar` are started on two cores operating in lock-step.
Then *one* of the following may happen:

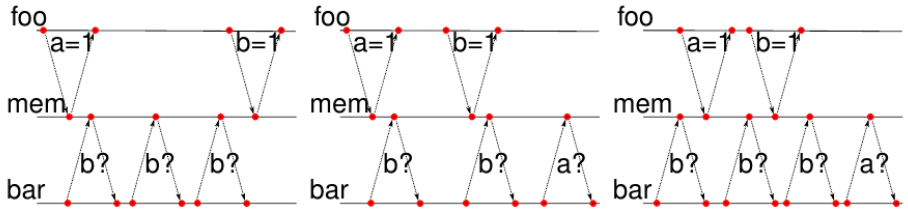


Unrealistic to assume that there is only one order between memory accesses:

- each conditional (and loop iteration) doubles the number of possible lock-step executions
- processors use caches ⇝ lock-step assumption is violated since cache behavior depends on data

Strict Consistency

Assuming `foo` and `bar` are started on two cores operating in lock-step.
Then *one* of the following may happen:



Unrealistic to assume that there is only one order between memory accesses:

- each conditional (and loop iteration) doubles the number of possible lock-step executions
- processors use caches ⇝ lock-step assumption is violated since cache behavior depends on data

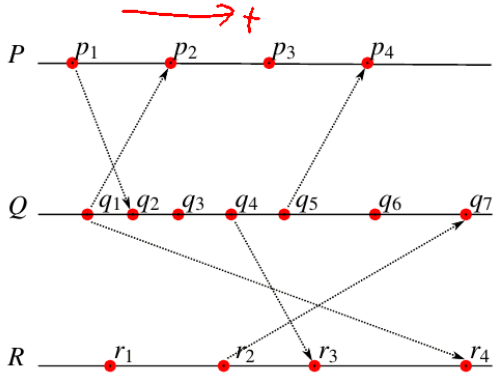
⇝ strict consistency is too strong to be realistic
⇝ state correctness in terms of what event *may* happen before another one

Events in a Distributed System

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

Events in a Distributed System

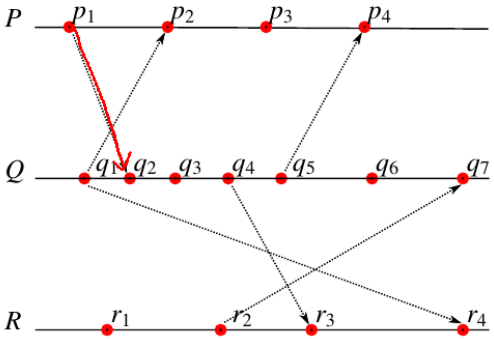
A process as a series of events [Lam78]: Given a distributed system of processes $P, \ldots$, each process P consists of events $p_1, p_2, \ldots$.
Example:



- event p_i in process P *happened before* p_{i+1}

Events in a Distributed System

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



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

Wand Law (I)

Events in time are like power of wands:

Wand Law (I)

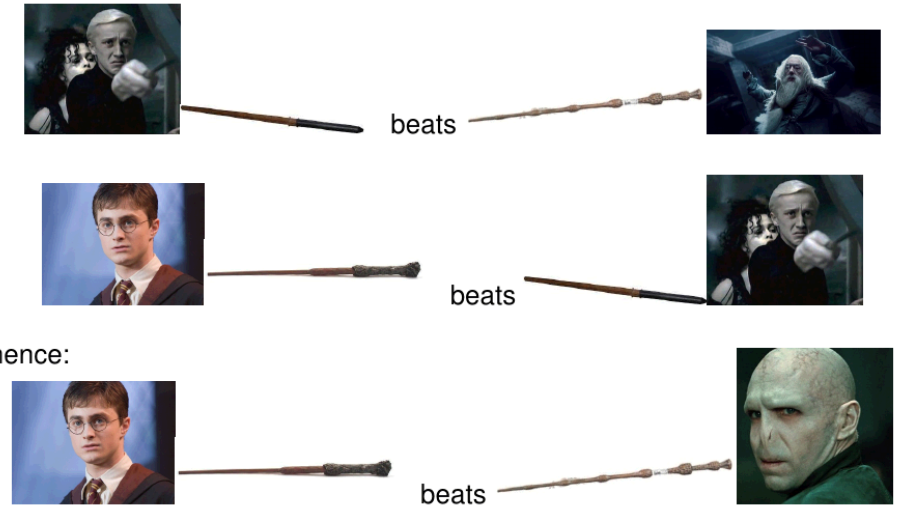
Events in time are like power of wands:

Wand Law (I)

Events in time are like power of wands:

Wand Law (I)

Events in time are like power of wands:



⇝ the "beats" relation is transitive

Wand Law (II)

More wand laws:

- "beats" is transitive
- "beats" is irreflexive

- implies that "beats" is asymmetric: if

then

⇝ "beats" is a *strict partial order*

The Happened-Before Relation

Definition

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

The Happened-Before Relation

Definition

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

Observe:

- $\to$ is partial (neither $p \to q$ or $q \to p$ may hold)

The Happened-Before Relation

Definition

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

Observe:

- $\to$ is partial (neither $p \to q$ or $q \to p$ may hold)
- $\to$ is irreflexive ($p \to p$ never holds)

The Happened-Before Relation

Definition

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

Observe:

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

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 $\neg(q \rightarrow p)$)

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 $\neg(q \rightarrow p)$)

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

Note: a strict partial order $\prec$ differs from a (non-strict) partial order $\preceq$ due to:

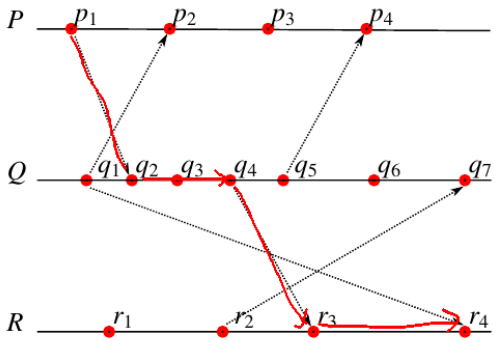
strict partial order	non-strict partial order
irreflexive $\neg(p \prec p)$	reflexive $p \preceq p$
asymmetric $p \prec q$ implies $\neg(q \prec p)$	antisymmetric $p \preceq q \wedge q \preceq p$ implies $p = q$

Concurrency

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 $q \not\rightarrow p$.



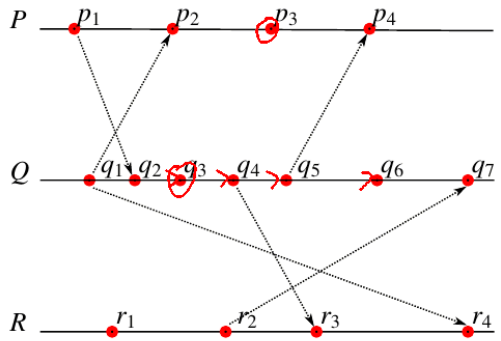
- $p_1 \rightarrow r_4$ in the example

Concurrency

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* that assigns a time-stamp $C(p)$ to each event p.

Definition (Clock Condition)

C satisfies the *clock condition* if for any events $p \to q$ then $C(p) < C(q)$.

Ordering

Let C be a *logical clock* that assigns a time-stamp $C(p)$ to each event p.

Definition (Clock Condition)

C satisfies the *clock condition* if for any events $p \to q$ then $C(p) < C(q)$.

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

1. if p_i and p_j are events of P and $p_i \to p_j$ then $C(p_i) < C(p_j)$
2. if 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* that assigns a time-stamp $C(p)$ to each event p.

Definition (Clock Condition)

C satisfies the *clock condition* if for any events $p \to q$ then $C(p) < C(q)$.

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

1. if p_i and p_j are events of P and $p_i \to p_j$ then $C(p_i) < C(p_j)$
2. if 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 is compatible with the strict partial order $\to$

Ordering

Let C be a *logical clock* that assigns a time-stamp $C(p)$ to each event p.

Definition (Clock Condition)

C satisfies the *clock condition* if for any events $p \to q$ then $C(p) < C(q)$.

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

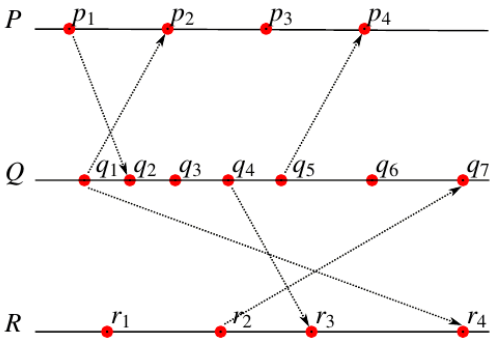
1. if p_i and p_j are events of P and $p_i \to p_j$ then $C(p_i) < C(p_j)$
2. if p is the sending of a message by process P and q is the reception of this message by process Q then $C(p) < C(q)$

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

The *set* defined by C that satisfies the clock condition are exactly the *set* of executions possible in the system.
$\rightsquigarrow$ use the process model and $\to$ to define better consistency model

Defining C Satisfying the Clock Condition

Given:



e	p_1	p_2	p_3	p_4
$C(e)$	1	3	4	14

e	q_1	q_2	q_3	q_4	q_5	q_6	q_7
$C(e)$	2	5	6	9	10	11	42

e	r_1	r_2	r_3	r_4
$C(e)$	7	8	12	13

Summary

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

Moving Away from Strong Consistency

Idea: use process 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

Moving Away from Strong Consistency

Idea: use process 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

We obtain a model for *sequential consistency*.

Definition: Sequential Consistency

Definition (Sequential Consistency Condition for Multi-Processors)

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Definition: Sequential Consistency

Definition (Sequential Consistency Condition for Multi-Processors)

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Given a result of a program with n threads on a SC system,

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$

Definition: Sequential Consistency

Definition (Sequential Consistency Condition for Multi-Processors)

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Given a result of a program with n threads on a SC system,

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
2. there exists a total order $\exists C \,.\, C(p_i^j) < C(p_k^l)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,

Definition: Sequential Consistency

Definition (Sequential Consistency Condition for Multi-Processors)

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Given a result of a program with n threads on a SC system,

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
2. there exists a total order $\exists C \,.\, C(p_i^j) < C(p_k^l)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,
3. such that this execution has the same result.

Definition: Sequential Consistency

Definition (Sequential Consistency Condition for Multi-Processors)

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Given a result of a program with n threads on a SC system,

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
2. there exists a total order $\exists C \,.\, C(p_i^j) < C(p_k^l)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,
3. such that this execution has the same result.

Yet, in other words:

- 1 defines the *execution path* of each thread

Definition: Sequential Consistency

Definition (Sequential Consistency Condition for Multi-Processors)

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Given a result of a program with n threads on a SC system,

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
2. there exists a total order $\exists C \,.\, C(p_i^j) < C(p_k^l)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,
3. such that this execution has the same result.

Yet, in other words:

- 1 defines the *execution path* of each thread
- the total order defined in 2 is one *interleaving* of the execution paths

Definition: Sequential Consistency

Definition (Sequential Consistency Condition for Multi-Processors)

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Given a result of a program with n threads on a SC system,

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
2. there exists a total order $\exists C \,.\, C(p_i^j) < C(p_k^l)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,
3. such that this execution has the same result.

Yet, in other words:

- 1 defines the *execution path* of each thread
- the total order defined in 2 is one *interleaving* of the execution paths
- 3 stipulates that the result of running the threads with this interleaving is always the same

Disproving Sequential Consistency

Given a result of a program with n threads on a SC system,

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
2. there exists a total order $\exists C \,.\, C(p_i^j) < C(p_k^l)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,
3. such that this execution has 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

Disproving Sequential Consistency

Given a result of a program with n threads on a SC system,

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
2. there exists a total order $\exists C \,.\, C(p_i^j) < C(p_k^l)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,
3. such that this execution has 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 $\rightarrow$

Disproving Sequential Consistency

Given a result of a program with n threads on a SC system,

1. with operations $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $\ldots p_0^n, p_1^n, \ldots$
2. there exists a total order $\exists C \,.\, C(p_i^j) < C(p_k^l)$ for all $i, j, k, l \ldots$ where $j = l$ implies $i < k$,
3. such that this execution has 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 differ

Memory Consistency | Sequential Consistency | 17 / 48

Weakening the Model

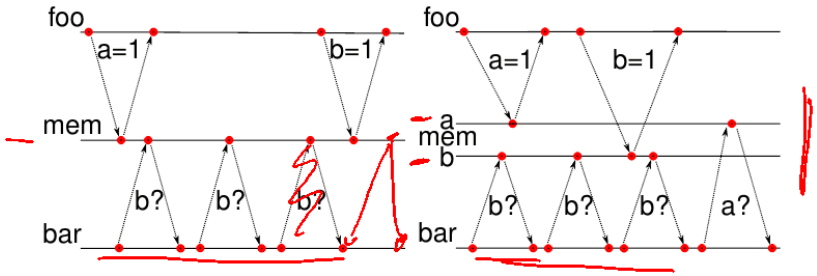
There is no observable change if calculations on different memory locations can happen in parallel.

- idea: model each memory location as a different process

Weakening the Model

There is no observable change if calculations on different memory locations can happen in parallel.

- idea: model each memory location as a different process

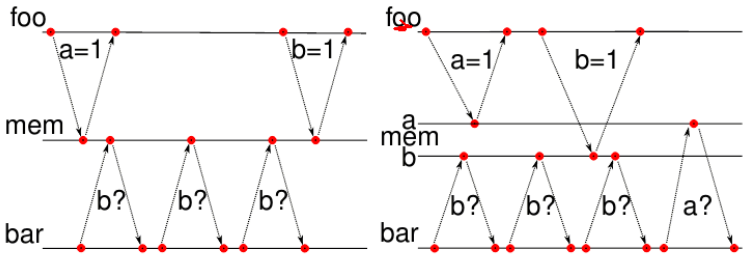


Memory Consistency | Sequential Consistency | 18 / 48

Weakening the Model

There is no observable change if calculations on different memory locations can happen in parallel.

- idea: model each memory location as a different process



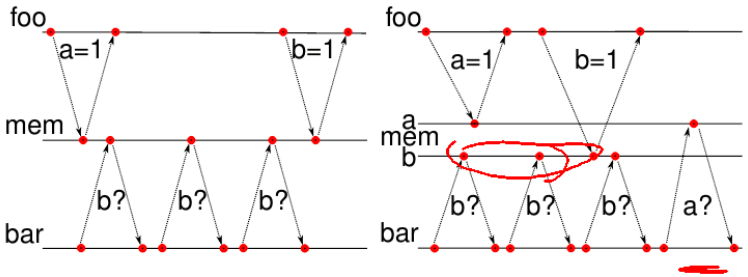
Sequential consistency still obeyed:

- the accesses of `foo` to `a` occurs before `b`

Weakening the Model

There is no observable change if calculations on different memory locations can happen in parallel.

- idea: model each memory location 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`

Benefits of Sequential Consistency

Benefits of the sequential consistency model:

- concisely represent *all* interleavings that are due to variations in speed
- synchronization using time is uncommon for software
- $\rightsquigarrow$ a good model for correct behaviors of concurrent programs
- $\rightsquigarrow$ programs results besides SC results are undesirable (they contain *races*)

Benefits of Sequential Consistency

Benefits of the sequential consistency model:

- concisely represent *all* interleavings that are due to variations in speed
- synchronization using time is uncommon for software
- $\rightsquigarrow$ a good model for correct behaviors of concurrent programs
- $\rightsquigarrow$ programs results besides SC results are undesirable (they contain *races*)

It is a realistic model for older hardware:

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

Benefits of Sequential Consistency

Benefits of the sequential consistency model:

- concisely represent *all* interleavings that are due to variations in speed
- synchronization using time is uncommon for software
- ⇝ a good model for correct behaviors of concurrent programs
- ⇝ programs results besides SC results are undesirable (they contain *races*)

It is a realistic model for older hardware:

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

Not a realistic model for modern hardware with out-of-order execution:

- what other processors see is determined by complex optimizations to caching

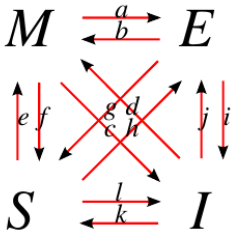
⇝ need to understand how caches work

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:



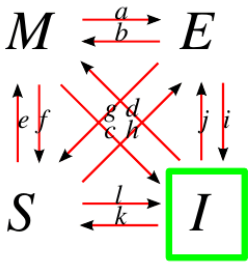
The MESI Protocol: States

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

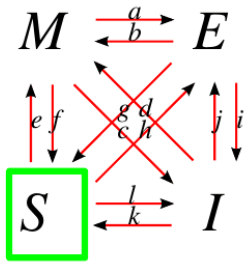
Processors (and also: GPUs, intelligent I/O devices) use caches to avoid a costly round-trip to RAM for every memory access.

- 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

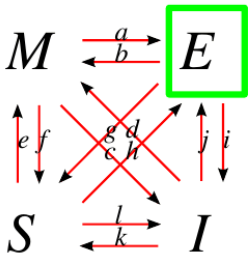
S: other caches have an identical copy of this cache line, it is *shared*



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



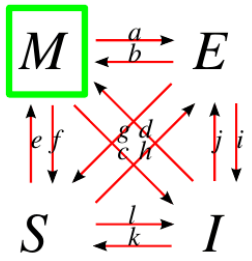
Each cache line is in one of the states M, E, S, I:

- 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



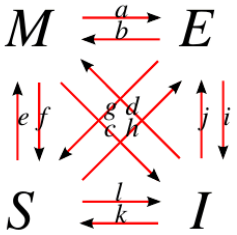
Each cache line is in one of the states M, E, S, I:

- 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:

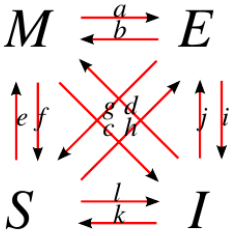
- I: it is *invalid* and is ready for re-use
- S: other caches have an identical copy of this cache line, it is *shared*
- E: the content is in no other cache; it is *exclusive* to this cache and can be overwritten without consulting other caches
- M: the content is exclusive to this cache and has furthermore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

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

The MESI Protocol: Messages

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

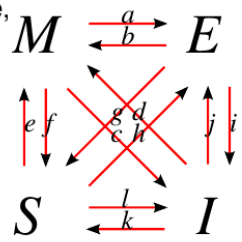
- *Read:* sent if CPU needs to read from an address



The MESI Protocol: Messages

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

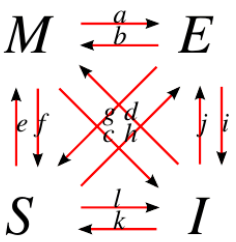
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address



The MESI Protocol: Messages

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

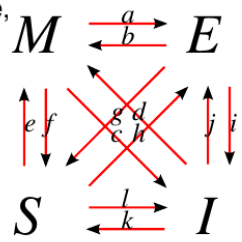
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address
- *Invalidate:* asks others to evict a cache line



The MESI Protocol: Messages

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

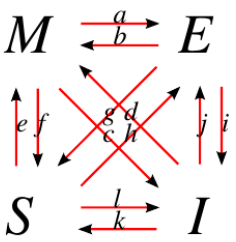
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address
- *Invalidate:* asks others to evict a cache line
- *Invalidate Acknowledge:* reply indicating that an address has been evicted



The MESI Protocol: Messages

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

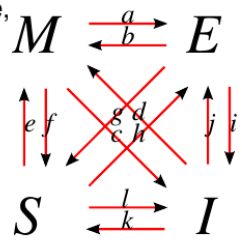
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address
- *Invalidate:* asks others to evict a cache line
- *Invalidate Acknowledge:* reply indicating that an address has been evicted
- *Read Invalidate:* like *Read*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



The MESI Protocol: Messages

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

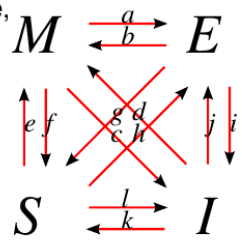
- *Read:* sent if CPU needs to read from an address
- *Read Response:* response to a *read* message, carries the data at the requested address
- *Invalidate:* asks others to evict a cache line
- *Invalidate Acknowledge:* reply indicating that an address has been evicted
- *Read Invalidate:* like *Read*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



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

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

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

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		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 (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		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
```

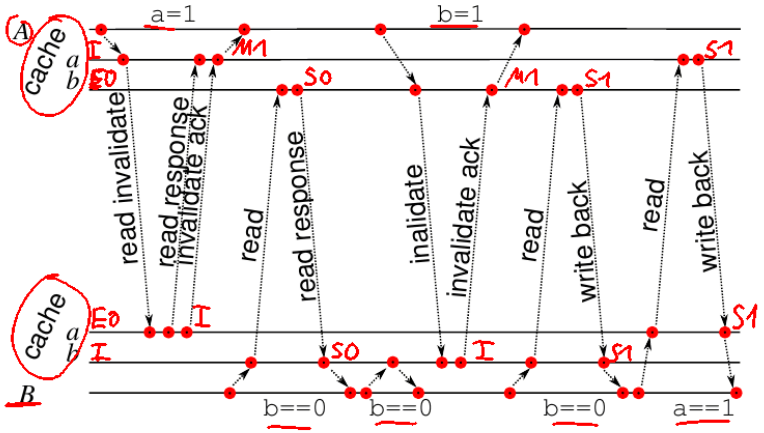
Thread B

```
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: Happened Before Model

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

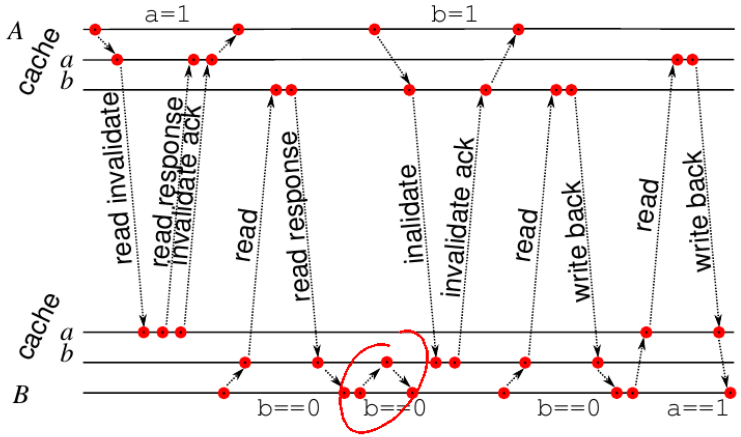


Observations:

- 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:

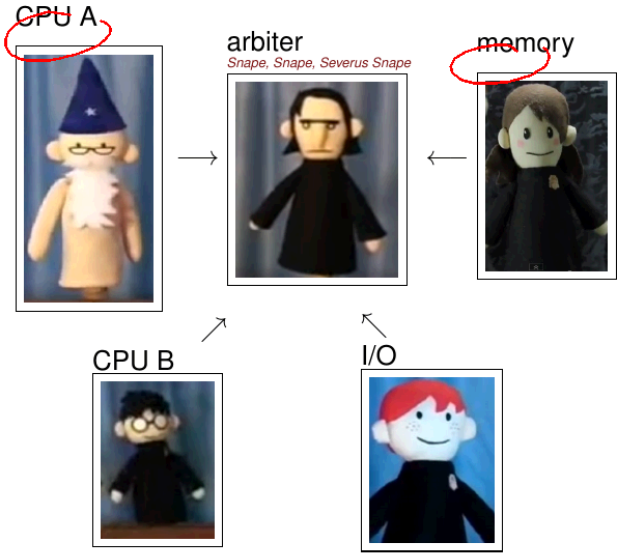
- each memory access must complete before executing next instruction
  ⇝ add edge
- second execution of test `b==0` stays within cache ⇝ no traffic

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?

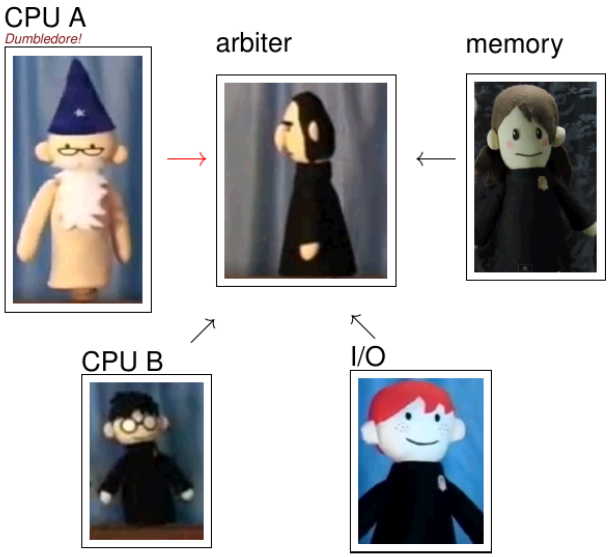
If two processors emit a message at the same time, the protocol might break.
Access to common bus is coordinated by an *arbiter*:



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"

Summary

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

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
```