Script **generated by TTT**

Title: Petter: Programmiersprachen (14.10.2015)

Date: Wed Oct 14 14:21:05 CEST 2015

Duration: 90:18 min

Pages: 49

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Dr. Michael Petter

Winter term 2015

Overall Structure

This course is given by

- Michael Petter: `petter@in.tum.de`
- there are at least 12 lectures
- there will be a written exam
- there is *no repeated* exam in this winter or the following summer semester

Overall Structure

This course is given by

- Michael Petter: `petter@in.tum.de`
- there are at least 12 lectures
- there will be a written exam
- there is *no repeated* exam in this winter or the following summer semester

We present selected topics that focus on current issues in the design or implementation of programming languages:

- concurrency
- modularization

Outline

1. Concurrency

1. Low-Level Concurrency, Memory Barriers
2. Wait-Free Algorithms
3. Locks and Monitors
4. Transactional Memory

2. Modularization

1. Method dispatching
2. Multiple Inheritance
3. Mixins and Traits
4. Prototype based Languages
5. Aspect Orientation
6. Generics and Templates

Administration

Lectures:

- every Wednesday at 14:15pm
- no lectures on bank holidays: 23.12., 30.12., 6.1.
- slides on `http://www2.in.tum.de/hp/Main?nid=281`
- lectures are recorded `http://ttt.in.tum.de`

Administration

Lectures:

- every Wednesday at 14:15pm
- no lectures on bank holidays: 23.12., 30.12., 6.1.
- slides on `http://www2.in.tum.de/hp/Main?nid=281`
- lectures are recorded `http://ttt.in.tum.de`

Exercises:

- an exercise sheet is made available with every lecture
- solving the sheets is *voluntary* but *recommended*

Administration

Lectures:

- every Wednesday at 14:15pm
- no lectures on bank holidays: 23.12., 30.12., 6.1.
- slides on `http://www2.in.tum.de/hp/Main?nid=281`
- lectures are recorded `http://ttt.in.tum.de`

Exercises:

- an exercise sheet is made available with every lecture
- solving the sheets is *voluntary* but *recommended*

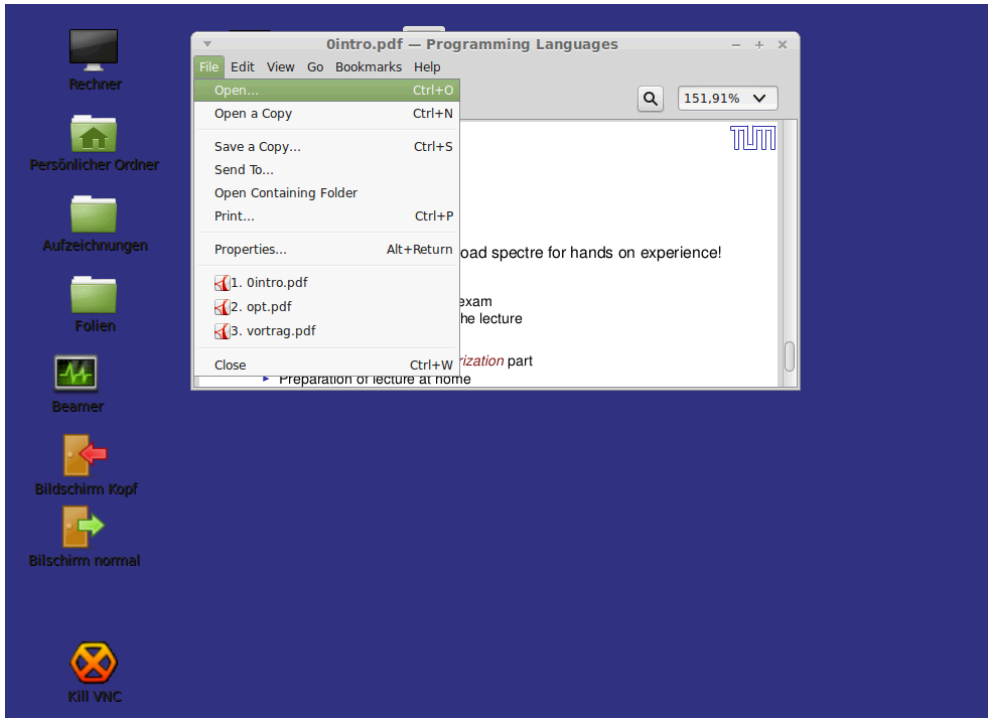
Tutorial sessions:

- for now, Fridays at 10.00am in this room (H2), starting *Oct. 16*
- the exercises are meant to be completed at home
- solutions to these exercises will be presented
- be there!
  - if you have questions of the general kind
  - if you have problems with the exercise sheet

Hands On!

Programming Languages offer a broad spectre for hands on experience!

1. Programming projects
   - Earn a .3 bonus on the final exam
   - Announced in the middle of the lecture
2. Inverted Classroom
   - Switch over to IC for *Modularization* part
   - Preparation of lecture at home
   - Hands on experiences live at the "lecture"



TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Memory Consistency

Dr. Michael Petter
Winter term 2015

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

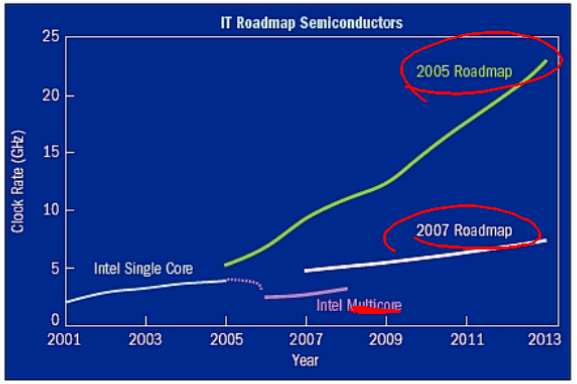
⇝ Intel could have built a processor with 256 Pentium cores in 2006

⚠ 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

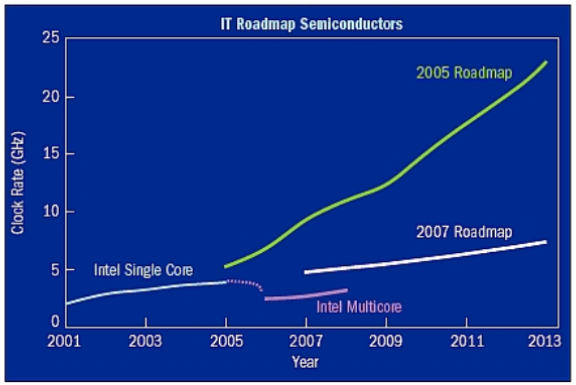
Single processors cannot be made much faster due to physical limitations.



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`
- 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 the code is *correct* if the `assert` holds

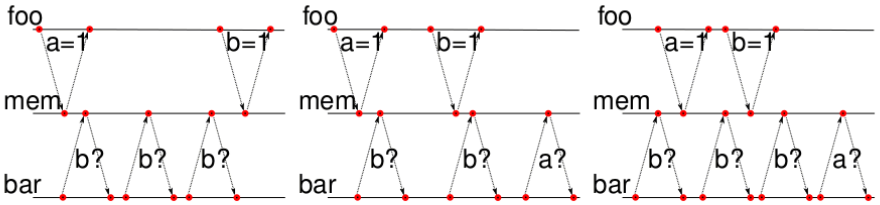
⇝ correctness means: writing a `1` to `a` *happens before* reading a `1` in `b`

Definition (Strict consistency)

Read operations from location l return values, written by the most recent write operation to l.

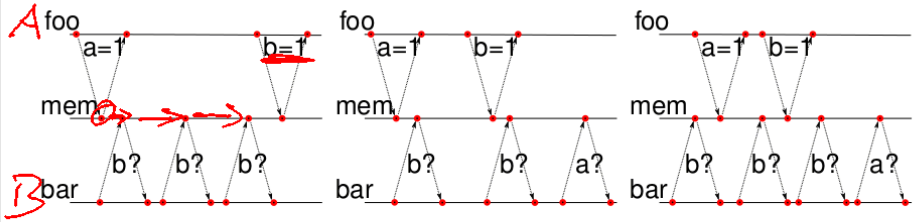
Strict Consistency

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



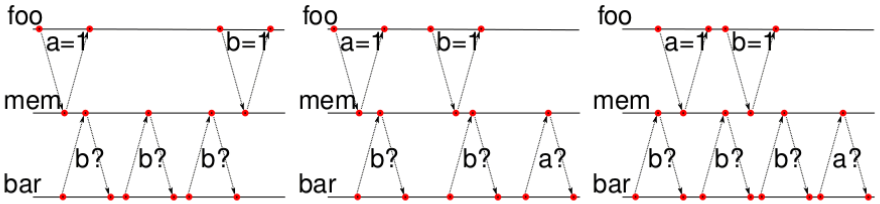
Strict Consistency

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



Strict Consistency

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

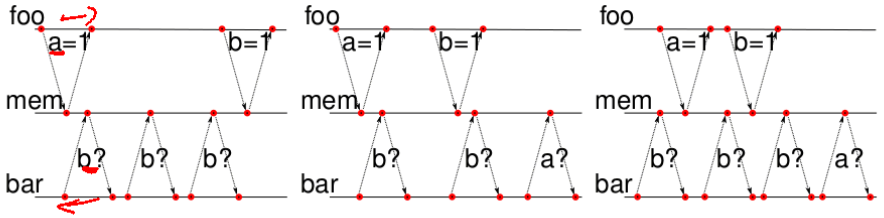


A unique order between memory accesses is unrealistic in reality:

- 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:



A unique order between memory accesses is unrealistic in reality:

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

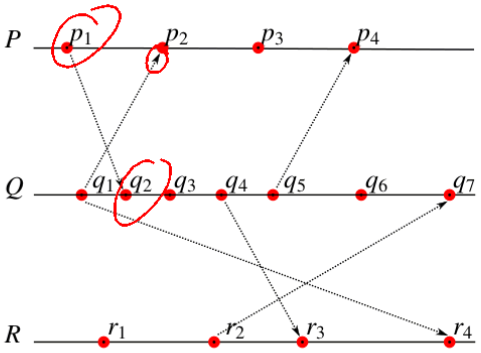
The Happend-Before Relation in Distributed Systems

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 $\bullet p_1, \bullet p_2, \ldots$.

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:



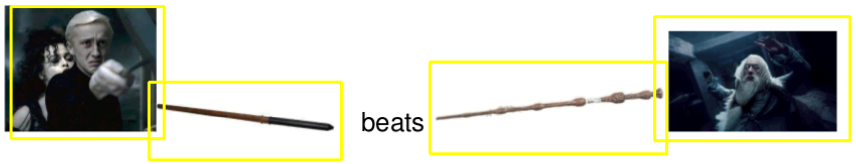
- event $\bullet p_i$ in process P *happened before* $\bullet p_{i+1}$
- if $\bullet p_i$ is an event that sends a message to Q then there is some event $\bullet q_j$ in Q that receives this message and $\bullet p_i$ *happened before* $\bullet q_j$

Excursion: Wandlore (I)

Events in time are like power of wands:

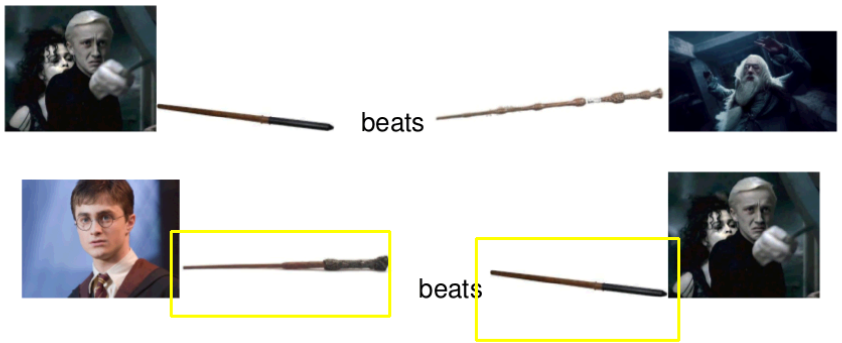
Excursion: Wandlore (I)

Events in time are like power of wands:



Excursion: Wandlore (I)

Events in time are like power of wands:



Excursion: Wandlore (I)

Events in time are like power of wands:

hence:

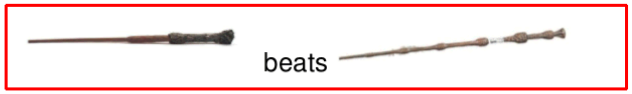
$\rightsquigarrow$ the "beats" relation is transitive

Excursion: Wandlore (II)

More wand laws:

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

- implies that "beats" is asymmetric: if



then

$\rightsquigarrow$ "beats" is a *strict partial order*

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 $\neg(q \rightarrow p)$)

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

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 $p \to r$)
- $\to$ is asymmetric (if $p \to q$ then $\neg(q \to p)$)

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

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

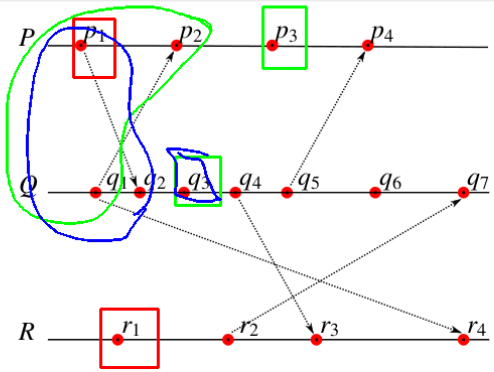
strict partial order	non-strict partial order
irreflexive $\neg(p \prec p)$	reflexive $p \preceq p$
asymmetric $p \prec q$ implies $\neg(q \prec p)$	antisymmetric $p \preceq q \wedge q \preceq p$ implies $p = q$

Concurrency in Process Diagrams

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

Definition

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



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

Ordering

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

Definition (Clock Condition)

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

$$p \to q \implies C(p) < C(q)$$

Ordering

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

Definition (Clock Condition)

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

$$p \to q \implies C(p) < C(q)$$

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* that assigns a 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* that assigns a 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:

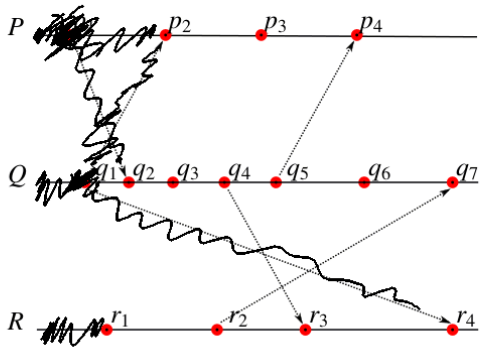
1. p_i and p_j are events of P and $p_i \rightarrow p_j$ then $C(p_i) < C(p_j)$
2. p is the sending of a message by process P and q is the reception of this message by process Q then $C(p) < C(q)$

$\rightsquigarrow$ 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 $\rightarrow$

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

Defining C Satisfying the Clock Condition

Given:



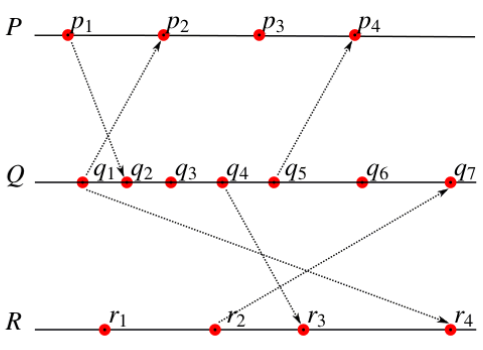
e	p_1	p_2	p_3	p_4
$C(e)$	1			

e	q_1	q_2	q_3	q_4	q_5	q_6	q_7
$C(e)$	2						

e	r_1	r_2	r_3	r_4
$C(e)$	3			

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

Sequential Consistency on Multi-Processor Machines

Moving Away from Strict 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

Definition: Sequential Consistency

Definition (Sequential Consistency Condition [Lam78])

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.

Sequential Consistency applied to Multiprocessor Programs:

Given a program with n threads,

1. for fixed operation 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 obey 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.

Disproving Sequential Consistency

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

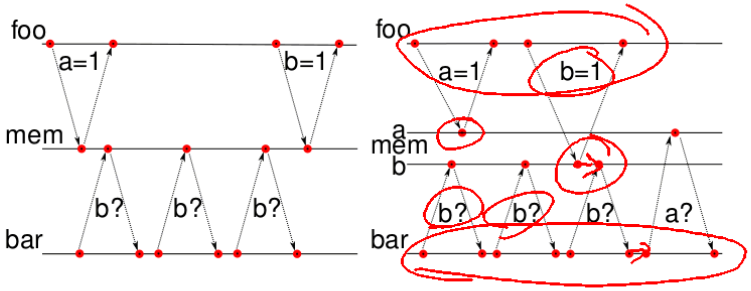
1. for fixed operation sequences $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $p_0^n, p_1^n, \ldots$ keeping the program order
2. executions obey the clock condition on the p_j^i,
3. all executions have the same result

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

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

Introducing Caches: The MESI Protocol