Script generated by TTT

Title: Petter: Programmiersprachen (19.10.2016)

Date: Wed Oct 19 14:48:05 CEST 2016

Duration: 67:19 min

Pages: 31

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

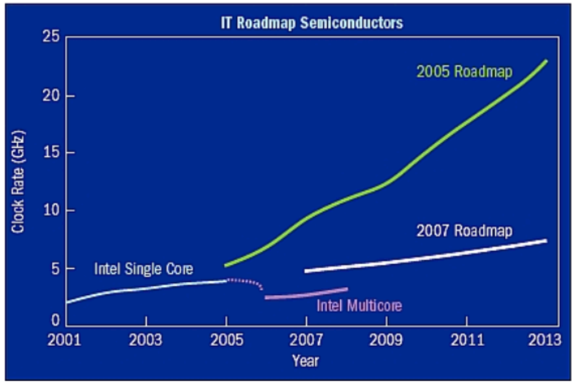
⇝ Intel could have built a processor with 256 Pentium cores in 2006

⚠ 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

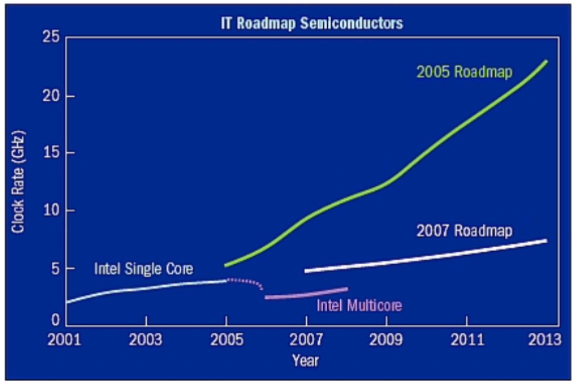
Single processors cannot be made much faster due to physical limitations.



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

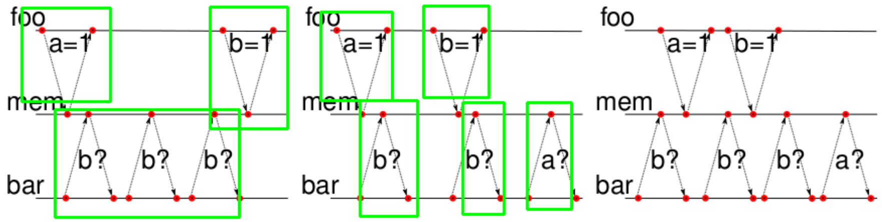
⇝ correctness means: writing `a` `1` to `a` *happens before* reading a `1` in `b`

Definition (Strict consistency)

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

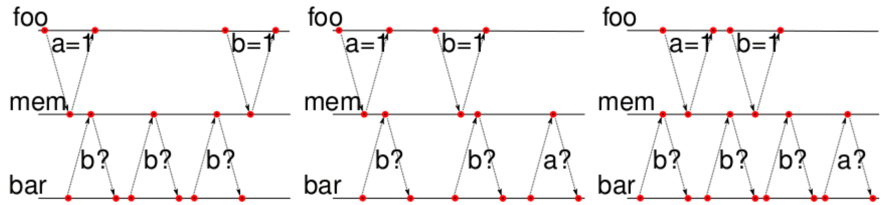
Strict Consistency

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:

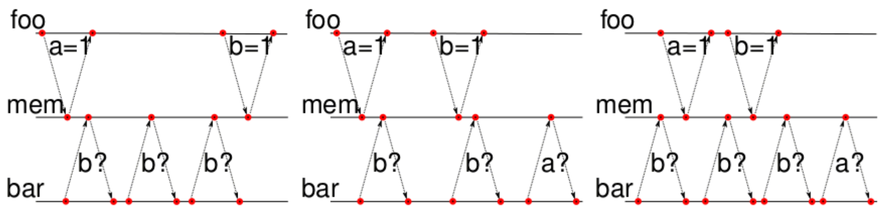


A unique order between memory accesses is unrealistic in reality:

- 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

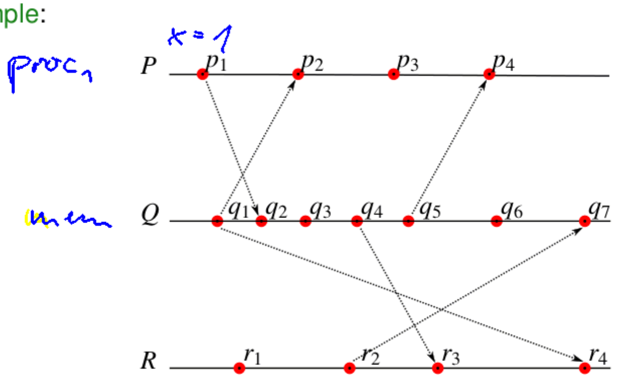
⇝ strict consistency is too strong to be realistic
Idea: state correctness in terms of what event *may* happen before another one

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* $\bullet q_j$

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

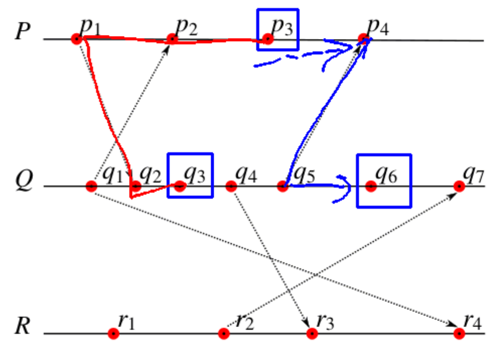
$\leadsto$ the $\rightarrow$ relation is a *strict partial order*

Concurrency in Process 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* 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

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

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

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

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

1. p_i and p_j are events of P and $p_i \rightarrow p_j$ 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 $C(p) < C(q)$

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

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

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

1. p_i and p_j are events of P and $p_i \rightarrow p_j$ 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 $\rightarrow$

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

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

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

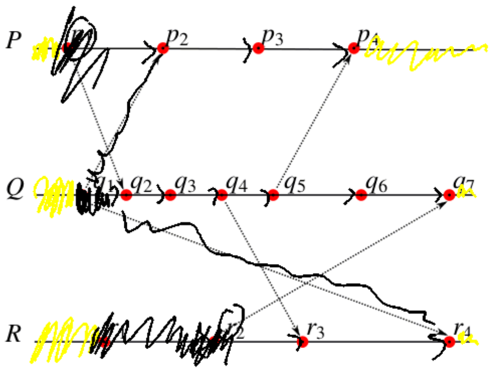
1. p_i and p_j are events of P and $p_i \rightarrow p_j$ then $C(p_i) < C(p_j)$
2. p is the sending of a message by process P and q is the reception of this message by process Q then $C(p) < C(q)$

$\leadsto$ 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:



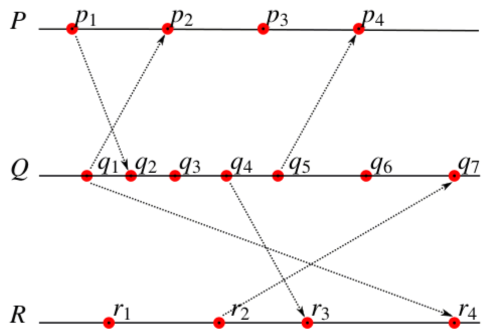
e	p_1	p_2	p_3	p_4
$C(e)$	2			

e	q_1	q_2	q_3	q_4	q_5	q_6	q_7
$C(e)$	1						

e	r_1	r_2	r_3	r_4
$C(e)$	0	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
- ⇝ concisely represent *some* interleavings

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
- ⇝ concisely represent *some* interleavings

We obtain a model for *sequential consistency*.

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,

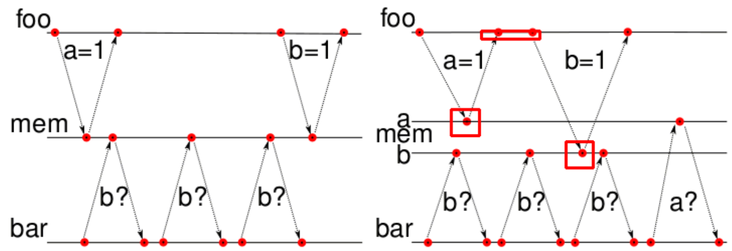
1. for fixed operation sequences $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $p_0^n, p_1^n, \ldots$ keeping the program order
2. executions obey the clock condition on the p_j^i,
3. all executions have the same result

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
- ⇝ a good model for correct behaviors of concurrent programs
- ⇝ 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
- ⇝ 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

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