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

2. Modularization
   - Method dispatching
   - Multiple Inheritance
   - Mixins and Traits
   - Prototype based Languages
   - Aspect Orientation
   - Generics and Templates

Administration

Lectures:
- every Wednesday at 14:15pm
- slides on http://ww2.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!

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

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

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

\[\text{\textbullet Intel could have built a processor with 256 Pentium cores in 2006}\]

\[\text{\textbullet However:}\]
- 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 \(m < n\) 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.

Concurrency for the Programmer

How is concurrency exposed in a programming language?
- spawning of new concurrent computations
- communication between threads

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?
- spawning of new concurrent computations
- communication between threads

*Communication* can happen in many ways:
- communication via shared memory *(this lecture)*
- atomic transactions on shared memory
- message passing

---

Communication between Cores

We consider the concurrent execution of these functions:

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

```c
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

**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:

```
(a) foo

b = ?
mem

(b) bar

b = ?
mem
```

---

Strict Consistency

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

```
(a) foo

a = ?
mem
```

```
(b) bar

b = ?
mem
```
Strict Consistency

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

$foo$

$\begin{align*}
&a = \top \\
&b = \top
\end{align*}$

$bar$

$\begin{align*}
&b? \\
&a
\end{align*}$

$mem$

$\begin{align*}
&b?
\end{align*}$

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 $\rightsquigarrow$ lock-step assumption is violated since cache behavior depends on data

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 $p_1, p_2, \ldots$
Events in a Distributed System

A process as a series of events [Lam78]: Given a distributed system of processes $P, Q, R, \ldots$, 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 $q_j$

Excursion: Wandlore (I)

Events in time are like power of wands:

- beats

Excursion: Wandlore (I)

Events in time are like power of wands:

- beats
**Excursion: Wandlore (I)**

Events in time are like power of wands:

```
beats
```

hence:

```
beats
```

--- the "beats" relation is transitive

**Excursion: Wandlore (II)**

More wand laws:
- "beats" is transitive
- "beats" is irreflexive

```
beats
```

implies that "beats" is asymmetric: if

```
beats
```
then

```
beats
```

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

--- the $\rightarrow$ relation is a **strict partial order**
The Happened-Before Relation

Definition
If an event $p$ happened before an event $q$ then $p \rightarrow q$.

Observe:
1. $\rightarrow$ is partial (neither $p \rightarrow q$ or $q \rightarrow p$ may hold)
2. $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
3. $\rightarrow$ is transitive ($p \rightarrow q \land q \rightarrow r$ then $p \rightarrow r$)
4. $\rightarrow$ is asymmetric (if $p \rightarrow q$ then $\neg(q \rightarrow p)$)

$\rightarrow$ the $\rightarrow$ relation is a strict partial order.

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

<table>
<thead>
<tr>
<th>strict partial order</th>
<th>non-strict partial order</th>
</tr>
</thead>
<tbody>
<tr>
<td>reflexive $\neg(p &lt; p)$</td>
<td>reflexive $p \leq p$</td>
</tr>
<tr>
<td>asymmetric $p &lt; q$ implies $\neg(q &lt; p)$</td>
<td>antisymmetric $p \leq q \land q \leq p$ implies $p = q$</td>
</tr>
</tbody>
</table>

Concurrency in Process Diagrams

Let $a \neq b$ abbreviate $\neg(a \rightarrow b)$.

Definition
Two distinct events $p$ and $q$ are said to be concurrent if $p \neq q$ and $q \neq p$.

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 $C(p) < C(q)$

$\rightarrow$ 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 $\rightarrow$

**Defining $C$ Satisfying the Clock Condition**

Given:

- $P, Q, R$ as processes
- $p_1, p_2, p_3, p_4$ as events in $P$
- $q_1, q_2, q_3, q_4, q_5, q_6, q_7$ as events in $Q$
- $r_1, r_2, r_3, r_4$ as events in $R$

<table>
<thead>
<tr>
<th>$C(e)$</th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$C(e)$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
<th>$q_5$</th>
<th>$q_6$</th>
<th>$q_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$C(e)$</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The set defined by all $C$ that satisfy the clock condition is exactly the set of executions possible in the system. $\rightarrow$ use the process model and $\rightarrow$ to define better consistency model.
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 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
- \( \sim \) 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,
- for fixed operation sequences \( p_0^1, p_1^1, \ldots \) and \( p_0^2, p_1^2, \ldots \) and \( p_0^i, p_1^i, \ldots \) keeping the program order
- executions obey the clock condition on the \( p_j^1 \),
- all executions have the same result
Yet, in other words:
- (6) defines the execution path of each thread
- each execution mentioned in (6) is one interleaving of processes
- (6) 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,
- for fixed operation sequences \( p_0^0, p_1^1, \ldots \) and \( p_0^0, p_1^1, \ldots \) keeping the program order
- executions obey the clock condition on the \( p_i^j \),
- 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 and a total ordering of all operations
- add extra processes to model other system components
- the original order becomes a partial order
- show that total orderings \( C' \) exist 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 \( \text{foo} \) 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)

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