Script generated by TTT

Title: Simon: Programmiersprachen (26.10.2012)
Date: Fri Oct 26 10:19:14 CEST 2012
Duration: 73:04 min
Pages: 70

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 \( \frac{1}{m} \approx n \) 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.

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

Learning Outcomes
- Happened-before Partial Order
- Sequential Consistency
- The MESI Cache Model
- Weak Consistency
- Memory Barriers
Communication between Cores

We consider the concurrent execution of these functions:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
</table>
| `void foo(void) {
    a = 1;
    b = 1;
} | `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:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
</table>
| `void foo(void) {
    a = 1;
    b = 1;
} | `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:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
</table>
| `void foo(void) {
    a = 1;
    b = 1;
} | `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
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:

```
foo
```

```
mem
```

```
bar
```

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 behaviour depends on data**

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 behaviour 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 [Lamport(1978)]: Given a distributed system of processes \( P_1, \ldots \), 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 \( q_j \).
The Happened-Before Relation

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

Observe:
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
- $\rightarrow$ is transitive ($p \rightarrow q$ and $q \rightarrow r$ then $p \rightarrow r$)
- $\rightarrow$ is asymmetric ($p \rightarrow q$ if and only if $\neg(q \rightarrow p)$)

The Happened-Before Relation

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

Observe:
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
- $\rightarrow$ is transitive ($p \rightarrow q$ and $q \rightarrow r$ then $p \rightarrow r$)
- $\rightarrow$ is asymmetric ($p \rightarrow q$ if and only if $\neg(q \rightarrow p)$)

\[ 5 \prec 6 \]

The Happened-Before Relation

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

Observe:
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
- $\rightarrow$ is transitive ($p \rightarrow q$ and $q \rightarrow r$ then $p \rightarrow r$)
- $\rightarrow$ is asymmetric ($p \rightarrow q$ if and only if $\neg(q \rightarrow p)$)

\[ 5 \prec 6 \]

The Happened-Before Relation

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

Observe:
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
- $\rightarrow$ is transitive ($p \rightarrow q$ and $q \rightarrow r$ then $p \rightarrow r$)
- $\rightarrow$ is asymmetric ($p \rightarrow q$ if and only if $\neg(q \rightarrow p)$)

\[ 5 \prec 6 \]

The Happened-Before Relation

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

Observe:
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
- $\rightarrow$ is transitive ($p \rightarrow q$ and $q \rightarrow r$ then $p \rightarrow r$)
- $\rightarrow$ is asymmetric ($p \rightarrow q$ if and only if $\neg(q \rightarrow p)$)

\[ 5 \prec 6 \]
The Happened-Before Relation

**Definition**
If an event \( p \) happened before an event \( q \) then \( p \rightarrow q \).

Observe:
- \( \rightarrow \) is irreflexive (\( p \rightarrow p \) never holds)
- \( \rightarrow \) is transitive (\( p \rightarrow q \land q \rightarrow r \) then \( p \rightarrow r \))
- \( \rightarrow \) is asymmetric (\( p \rightarrow q \) if and only if \( \neg (q \rightarrow p) \))
- the \( \rightarrow \) relation is a strict partial order

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

<table>
<thead>
<tr>
<th>strict partial order</th>
<th>non-strict partial order</th>
</tr>
</thead>
<tbody>
<tr>
<td>irreflexive ( \neg (p \rightarrow p) )</td>
<td>reflexive ( p \preceq p )</td>
</tr>
<tr>
<td>asymmetric ( p \prec q \iff \neg (q \preceq p) )</td>
<td>antisymmetric ( p \preceq q \land q \preceq p \Rightarrow p = q )</td>
</tr>
</tbody>
</table>

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 \( q \not\rightarrow p \).

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 \( p \rightarrow q \) 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 $p \rightarrow q$ then $C(p) < C(q)$.

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

1. if $p_1$ and $p_2$ are events of $P$ and $p_1 \rightarrow p_2$ then $C(p_1) < C(p_2)$.
2. if $p$ is the sending of a message by process $P$ and $q$ is the reception of this message by process $Q$ then $C(p) < C(q)$.

$\leadsto$ 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 $\rightarrow$.

---

**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 $p \rightarrow q$ then $C(p) < C(q)$.

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

1. if $p_1$ and $p_2$ are events of $P$ and $p_1 \rightarrow p_2$ then $C(p_1) < C(p_2)$.
2. if $p$ is the sending of a message by process $P$ and $q$ is the reception of this message by process $Q$ then $C(p) < C(q)$.

$\leadsto$ 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 $\rightarrow$.

---

**Sequential Consistency**

Note: there is no observable change if calculations on different memory locations can happen in parallel.

- model each memory location as different process

---

**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 $p \rightarrow q$ then $C(p) < C(q)$.

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

1. if $p_1$ and $p_2$ are events of $P$ and $p_1 \rightarrow p_2$ then $C(p_1) < C(p_2)$.
2. if $p$ is the sending of a message by process $P$ and $q$ is the reception of this message by process $Q$ then $C(p) < C(q)$.

$\leadsto$ 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 $\rightarrow$.

The set of $C$ that satisfy the clock condition are exactly the set of executions possible in the system.

$\leadsto$ use the process model and $\rightarrow$ to define better consistency model

---

**Sequential Consistency**

Note: there is no observable change if calculations on different memory locations can happen in parallel.

- model each memory location as different process
Sequential Consistency

Note: there is no observable change if calculations on different memory locations can happen in parallel.

- model each memory location as different process

Some observations:
- the accesses of foo to a occurs before b

Sequential Consistency

Note: there is no observable change if calculations on different memory locations can happen in parallel.

- model each memory location as different process

Some observations:
- 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:
- it is a realistic model for well-behaved concurrency
- ... since it allows to concisely represent some interleavings
**Sequential Consistency**

Note: there is no observable change if calculations on different memory locations can happen in parallel.

- model each memory location as different process

```
foo
  a=1
  b=1

mem
  a
  b
  bar
```

Some observations:

- 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:

- it is a realistic model for well-behaved concurrency
- ... since it allows to concisely represent *some* interleavings

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* that avoid the exclusive access to memory as much as possible

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

- memory accesses are only *sequentially consistent* from the point of view of the processor executing them
- what other processors see is determined by the complex protocol of the caches
- 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:

\[
\begin{align*}
M & \xrightarrow{a} E \\
S & \xrightarrow{b} I
\end{align*}
\]

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

~ the state of cache lines is kept consistent by sending messages

The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenney(2010)]:

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

Moving data between caches is coordinated by sending messages [McKenney(2010)]:

- 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 [McKenny(2010)]:

- **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

\[
\begin{array}{c}
M \xrightarrow{b} E \\
i & \downarrow & \uparrow \\
S & \xrightarrow{i} I
\end{array}
\]

- **Writeback**: info on what data has been sent to main memory
The MESI Protocol: Messages

Moving data between caches is coordinated by sending messages [McKenny(2010)]:

- **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

Additional **store** and **read** messages are transmitted to main memory.

---

MESI Example (I)

**Thread A**

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

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

**Thread B**

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

<table>
<thead>
<tr>
<th>statement</th>
<th>CPU A</th>
<th>CPU B</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>1M</td>
<td>1M</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1M</td>
<td>1M</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1M</td>
<td>1S</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1M</td>
<td>1S</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>read invalidate of a from CPU A</td>
</tr>
<tr>
<td>invalidate ack of a from CPU B</td>
</tr>
<tr>
<td>read response of b=1 from CPU A</td>
</tr>
<tr>
<td>read of a from CPU B</td>
</tr>
<tr>
<td>read response of a=1 from CPU A</td>
</tr>
</tbody>
</table>

---

MESI Example (II)

**Thread A**

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

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

**Thread B**

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

<table>
<thead>
<tr>
<th>statement</th>
<th>CPU A</th>
<th>CPU B</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>1M</td>
<td>1M</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1M</td>
<td>1M</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1M</td>
<td>1S</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1M</td>
<td>1S</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>read invalidate of b from CPU B</td>
</tr>
<tr>
<td>read response of b=1 from CPU A</td>
</tr>
<tr>
<td>read of a from CPU A</td>
</tr>
<tr>
<td>read response of a=1 from CPU A</td>
</tr>
</tbody>
</table>

---

MESI Example: Happened Before Model

*Idea*: each cache line one process, model each message

**Observations:**
- each memory access must complete before executing next instruction
- add edge
**MESI Example: Happened Before Model**

*Idea:* each cache line one process, model each message

**Observations:**
- Each memory access must complete before executing next instruction.
  - `add edge`
- Second execution of test `b==0` stays within cache `→` no traffic.

**Store Buffers**

*Goal:* continue execution after write operation

- Put each write into a **store buffer** and trigger reception of cache line.

---

**MESI Example: Happened Before Model**

*Idea:* each cache line one process, model each message

**Observations:**
- Each memory access must complete before executing next instruction.
  - `add edge`
- Second execution of test `b==0` stays within cache `→` no traffic.
- Writes always stall `→` CPU A should continue executing after `a = 1`.

**Store Buffers**

*Goal:* continue execution after write operation

- Put each write into a **store buffer** and trigger reception of cache line.
- Once a cache line has arrived, apply relevant writes.
Store Buffers

Goal: continue execution after write operation

- put each write into a store buffer and trigger reception of cache line
- once a cache line has arrived, apply relevant writes
  - store buffer is a set
  - △ sequential consistency per CPU is violated unless
    - each read checks store buffer before cache

What about sequential consistency for the whole system?
Happened-Before Model for Store Buffers

Thread A
\[
a = 1; \\
b = 1;
\]

Thread B
\[
\begin{align*}
\text{while } (b == 0) \{ & \\
\text{assert}(a == 1); & \\
\}
\end{align*}
\]

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I

Explicit Synchronisation: 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
Explicit Synchronisation: 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
- Intel x86 CPUs provide the sfence instruction

Happened-Before Model for Write Fences

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