Graduate Seminar on Weakly Consistent Concurrency

Ori Lahav

TEL AVIV UNIVERSITY אוניברסיטת תל אביב

March 7, 2018

Graduate seminar on weakly consistent concurrency

Today:

- What is this seminar about?
- Requirements and logistics for the seminar
- Introduction to a weak concurrency semantics (simplified C11)

Semantics of concurrent systems

- We expect concurrent systems to guarantee strong consistency:
  - Operations of different components can be arranged on a global timeline
  - Each operation is aware of all previous operations
  - An illusion of a centralized system
- This requirement severely limits parallelism:
  - performance
  - availability and fault tolerance

Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

```
$x := 1$;                       ||    $y := 1$;
$a := y$;                       ||    $b := x$;
**if** $(a = 0)$ **then**       ||    **if** $(b = 0)$ **then**
  /* critical section */        ||      /* critical section */
```

Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

```
x := 1;                        ||   y := 1;
a := y;                        ||   b := x;
if (a = 0) then                ||   if (b = 0) then
  /* critical section */       ||     /* critical section */
```

Is it safe?

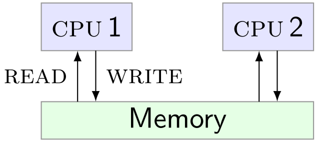
Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

```
    $x := 1;$                          ||    $y := 1;$
    $a := y;$  // 0                    ||    $b := x;$  // 0
    if $(a = 0)$ then                  ||    if $(b = 0)$ then
      /* critical section */           ||      /* critical section */
```

Is it safe?

Yes, if we assume sequential consistency (SC):



Example: Shared-memory concurrency in C++

```
int X, Y, a, b;

void thread1() {
    X = 1;
    a = Y;
}

void thread2() {
    Y = 1;
    b = X;
}
```

```
int main () {
    int cnt = 0;

    do {
        X = 0; Y = 0;

        thread first(thread1);
        thread second(thread2);

        first.join();
        second.join();
        cnt++;

    } while (a != 0 || b != 0);

    printf("%d\n",cnt);
    return 0;
}
```

Example: Shared-memory concurrency in C++

```
int X, Y, a, b;

void thread1() {
    X = 1;
    a = Y;
}

void thread2() {
    Y = 1;
    b = X;
}
```

```
int main () {
    int cnt = 0;

    do {
        X = 0; Y = 0;

        thread first(thread1);
        thread second(thread2);

        first.join();
        second.join();
        cnt++;

    } while (a != 0 || b != 0);

    printf("%d\n",cnt);
    return 0;
}
```

If Dekker's mutual exclusion is safe, this program will not terminate

Sequentially consistent shared memory?

No existing hardware implements SC!

- SC is very expensive (memory $\sim$100 times slower than CPU).
- SC does not scale to many processors.
- SC forbids various useful (compiler) optimizations.

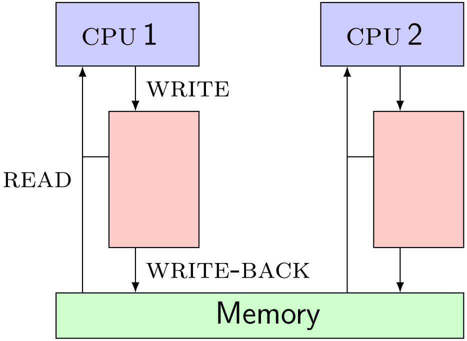
Instead, we have a weak memory model

- Allows more behaviors than SC (*weak behaviors* / *anomalies*)

Example: store buffering in x86-TSO

Initially, $x = y = 0$.

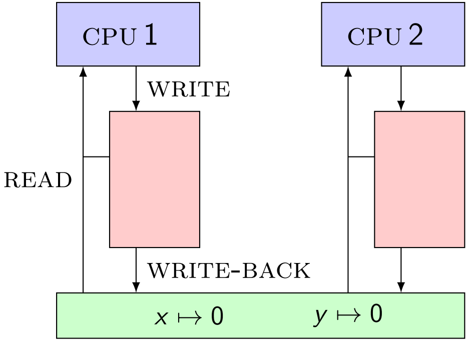
$x := 1;$	$y := 1;$
$a := y;$ // 0	$b := x;$ // 0



Example: store buffering in x86-TSO

Initially, $x = y = 0$.

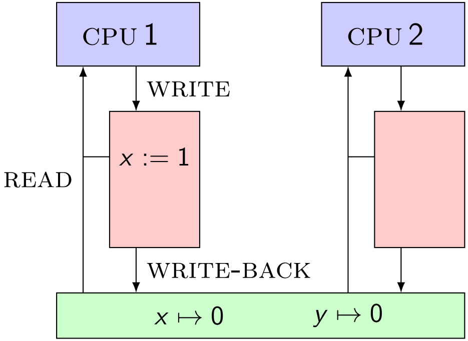
▶ $x := 1;$	▶ $y := 1;$
$a := y;$ // 0	$b := x;$ // 0



Example: store buffering in x86-TSO

Initially, $x = y = 0$.

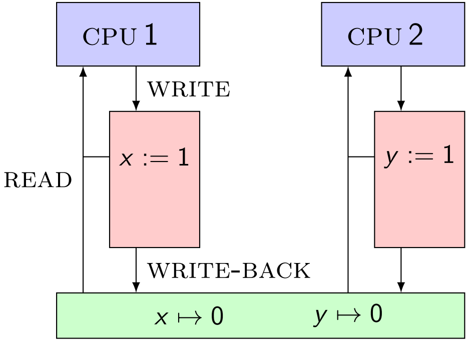
$x := 1;$	∥	▶ $y := 1;$
▶ $a := y;$ // 0	∥	$b := x;$ // 0



Example: store buffering in x86-TSO

Initially, $x = y = 0$.

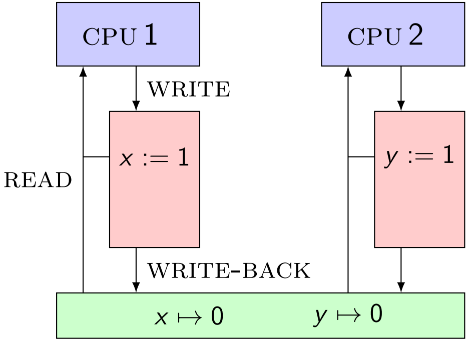
$x := 1;$	$y := 1;$
▶ $a := y;$ // 0	▶ $b := x;$ // 0



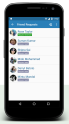
Example: store buffering in x86-TSO

Initially, $x = y = 0$.

$x := 1;$	$y := 1;$
fence;	**fence**;
$a := y;$ // 0	$b := x;$ // 0



Beyond shared memory concurrency



Initially, $x = y = 0$.

$$\begin{array}{l||l} x := 1; & a := y; \ \text{// 1} \\ y := 1; & b := x; \ \text{// 0} \end{array}$$

ARM allows this behavior.

Beyond shared memory concurrency

$$\begin{array}{l||l} \mathit{Email} := \text{“}\mathit{Dear\ Bob,...}\text{”}; & a := \mathit{Sms};\ \ /\!/\ \text{“}\mathit{check\ ur\ mail}\text{”} \\ \mathit{Sms} := \text{“}\mathit{check\ ur\ mail}\text{”}; & b := \mathit{Email};\ \ /\!/\ \text{“}\mathit{no\ new\ mail!}\text{”} \end{array}$$

Distributed systems: The CAP theorem

Theorem

A distributed data store to cannot simultaneously provide consistency, availability, and partition tolerance.

- Consistency: "any read operation that begins after a write operation completes must return that value, or the result of a later write operation".
- Availability: "every request received by a non-failing node in the system must result in a response".
- Partition tolerance: "the network will be allowed to lose arbitrarily many messages sent from one node to another".

[Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services]

Performance?

- Low latency: “if a server can safely respond to a user’s request when it is partitioned from all other servers, then it can also respond to requests without contacting other servers even when it is able to do so”.

Distributed databases: Serializability

Serializability: all possible outcomes can be explained by a serial schedule (i.e., sequential with no transaction overlap).

Example

T1:

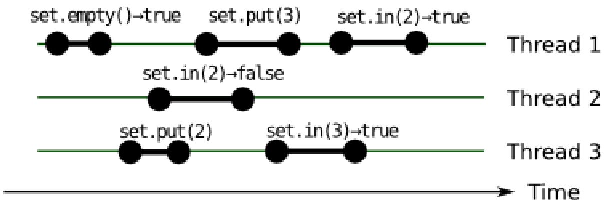
$$\left[\mathit{account}1 := \mathit{account}1 + 100\$ \right. \quad \Bigg\| \quad$$

T2:

$$\left[\begin{array}{l} \mathit{account}1 := \mathit{account}1 + 10\% \\ \mathit{account}2 := \mathit{account}2 + 10\% \end{array} \right.$$

- Serializability is a major correctness criterion (or: consistency guarantee) for distributed databases.
- Often it is too strong: commercial databases provide concurrency control with a whole range of consistency levels.

Adding real-time: Linearizability



- Linearizability is the major correctness criterion (or: consistency guarantee) for concurrent data structures.
- Often it is too strong, and we have to settle for *"eventual consistency"*: "updates in one component will eventually appear at other components, and if all components have received the same set of updates, they will have the same state".

(figure by Paul Shved, A Foo Walks into a Bar... `http://coldattic.info/info/`)

What is this seminar about?

Strong consistency is often too strong, and has to be replaced by weaker consistency guarantees.

Some questions:

- How to precisely specify weakly consistent semantics?
- How can we show that implementations meet certain weakly consistent specifications?
- How can we reason about weakly consistent concurrent systems?
- How can the users avoid weak behaviors in a given weakly concurrent system?

Why do we even need formal semantics?

- Informal partial specifications are often good enough to build useful systems and program in them
- But not to:
  - state and prove correctness of an implementation
  - reason about programs (formal verification)
  - compare different models
  - compiler correctness
  - establish robustness (a certain program does not have weak behaviors)

Requirements and logistics

Requirements

- Attend all meetings.
- Present one paper (70-80 minute talk).
- Prepare slides and/or handouts (pdf, in English), and send them to me a week before the lecture.
- The material will be publicly available.
- May work in pairs (present 2 papers).
- Discuss presentations with me before (e.g., on Sunday afternoon or the week before).
- Grade: meeting these requirements; understanding of the paper; quality and clarity of presentation in class; quality of slides/handouts.

Your presentations

- This is an advanced seminar: papers are not easy and not self-contained.
- Understand and present the crux, rather than all details.
- Demonstrate with clear examples.
- Be precise.
- May (and often should) skip proof details.

Logistics

- Website:
  http://www.cs.tau.ac.il/~orilahav/seminar18/
- By next week:
  paper assignments, add your preferences in
  https://docs.google.com/document/d/16LvQdzA_onDbli-OZOGR_5H8mVrjun8WBJlkettpLMk/edit?usp=sharing
- Speaker for next week?