IMP with Parallel Composition

Here’s a simple model of shared-memory parallelism: let’s extend IMP with a new a parallel composition command.
IMP with Parallel Composition

Here’s a simple model of shared-memory parallelism: let’s extend IMP with a new a parallel composition command.

\[
\begin{align*}
a & ::= x \mid n \mid a_1 + a_2 \\
b & ::= \text{true} \mid \text{false} \mid a_1 < a_2 \\
c & ::= \text{skip} \mid x := a \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \\
& \quad \mid c_1 \parallel c_2
\end{align*}
\]
Operational Semantics

And add small-step operational semantics rules for $c_1 \parallel c_2$ that interleave the execution of $c_1$ and $c_2$:

$$
\langle \sigma, c_1 \rangle \rightarrow \langle \sigma', c'_1 \rangle \\
\langle \sigma, c_1 \parallel c_2 \rangle \rightarrow \langle \sigma', c'_1 \parallel c_2 \rangle
$$
And add small-step operational semantics rules for $c_1 \parallel c_2$ that interleave the execution of $c_1$ and $c_2$:

$$\langle \sigma, c_1 \rangle \rightarrow \langle \sigma', c_1' \rangle$$

$$\langle \sigma, c_1 \parallel c_2 \rangle \rightarrow \langle \sigma', c_1 \parallel c_2 \rangle$$

$$\langle \sigma, c_2 \rangle \rightarrow \langle \sigma'.c_2' \rangle$$

$$\langle \sigma, c_1 \parallel c_2 \rangle \rightarrow \langle \sigma', c_1 \parallel c_2' \rangle$$
And add small-step operational semantics rules for \( c_1 \parallel c_2 \) that interleave the execution of \( c_1 \) and \( c_2 \):

\[
\begin{align*}
\langle \sigma, c_1 \rangle & \rightarrow \langle \sigma', c_1' \rangle \\
\langle \sigma, c_1 || c_2 \rangle & \rightarrow \langle \sigma', c_1 || c_2 \rangle \\
\langle \sigma, c_2 \rangle & \rightarrow \langle \sigma', c_2' \rangle \\
\langle \sigma, c_1 || c_2 \rangle & \rightarrow \langle \sigma', c_1 || c_2' \rangle \\
\langle \sigma, \text{skip} || \text{skip} \rangle & \rightarrow \langle \sigma, \text{skip} \rangle
\end{align*}
\]
And add small-step operational semantics rules for $c_1 \parallel c_2$ that interleave the execution of $c_1$ and $c_2$:

\[
\begin{align*}
\langle \sigma, c_1 \rangle &\rightarrow \langle \sigma', c_1' \rangle \\
\langle \sigma, c_1 \parallel c_2 \rangle &\rightarrow \langle \sigma', c_1' \parallel c_2 \rangle \\
\langle \sigma, c_2 \rangle &\rightarrow \langle \sigma'.c_2' \rangle \\
\langle \sigma, c_1 \parallel c_2 \rangle &\rightarrow \langle \sigma', c_1 \parallel c_2' \rangle \\
\langle \sigma, \text{skip} \parallel \text{skip} \rangle &\rightarrow \langle \sigma, \text{skip} \rangle
\end{align*}
\]

The rules allow either sub-command to take a step; two sub-commands can interleave read and write operations involving the same store.
Parallel Bank Account

What happens if we deposit into a bank account twice under parallel composition?

\[
\text{bal} := 0;
\quad (\text{bal} := \text{bal} + 21.0 \quad || \quad \text{bal} := \text{bal} + 21.0)
\]
Synchronization

Languages have synchronization constructs that control the interactions between threads.

Many languages have mutual exclusion, a.k.a. locking:

```plaintext
lock l;
bal := bal + 21.0;
unlock l
```
Synchronization

Languages have synchronization constructs that control the interactions between threads.

Many languages have mutual exclusion, a.k.a. locking:

```
lock l;
bal := bal + 21.0;
unlock l
```

A well-behaved alternative is transactional memory:

```
transaction {
    bal := bal + 21.0
}
```
Reasoning About Shared Memory

This program reads and writes two shared variables from two different “threads”:

\[ x := 0; y := 0; \]
\[ (y := 1; \text{tmp1} := x) \quad || \quad (x := 1; \text{tmp2} := y) \]

What can tmp1 and tmp2 be afterward?
$x := 0$

$y := 0$

$y := 1$

$x := 1$

$\text{tmp1} := x$

$\text{tmp2} := y$

\text{print tmp1 and tmp2}
The *happens before* relation is a partial order on *events* in a program execution.

See also Lamport, 1978: “Time, Clocks and the Ordering of Events in a Distributed System.”
The *happens before* relation is a partial order on *events* in a program execution.

Operation *a* happens before *b*, written *a* \(\rightarrow\) *b*, iff:

- *a* and *b* belong to the same thread and *a* comes before *b* in a single-threaded execution, or
- *a* sends an inter-thread message that *b* receives.

See also Lamport, 1978: “Time, Clocks and the Ordering of Events in a Distributed System.”
Happens Before

The *happens before* relation is a partial order on *events* in a program execution.

Operation $a$ happens before $b$, written $a \rightarrow b$, iff:

- $a$ and $b$ belong to the same thread and $a$ comes before $b$ in a single-threaded execution, or
- $a$ sends an inter-thread message that $b$ receives.

(Also add transitivity: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.)

See also Lamport, 1978: “Time, Clocks and the Ordering of Events in a Distributed System.”
Happens Before

In modern multithreaded programming, messages are sent and received at *synchronization* events:

- unlock \( l \) → lock \( l \)
- fork \( t \) → first operation in thread \( t \)
- last operation in thread \( t \) → join \( t \)
Legal Executions

Which executions of a multi-threaded program are possible?
Legal Executions

Which executions of a multi-threaded program are possible?

Model an execution as a total order $a \rightarrow_e b$ on the same set of events. For example:

```
y := 1  →  tmp1 := x  →  x := 1  →  tmp2 := y
```
Legal Executions

Which executions of a multi-threaded program are possible?

Model an execution as a total order $a \rightarrow_e b$ on the same set of events. For example:

$$y := 1 \rightarrow \text{tmp1 := x} \rightarrow x := 1 \rightarrow \text{tmp2 := y}$$

Then ask: is $\rightarrow \subseteq \rightarrow_e$? If so, then we say that $\rightarrow_e$ is a sequentially consistent execution.
Legal Executions

Which executions of a multi-threaded program are possible?

Model an execution as a total order \( a \rightarrow_e b \) on the same set of events. For example:

\[
\begin{align*}
y &:= 1 \\
tmp1 &:= x \\
x &:= 1 \\
tmp2 &:= y
\end{align*}
\]

Then ask: is \( \rightarrow \subseteq \rightarrow_e \)? If so, then we say that \( \rightarrow_e \) is a **sequentially consistent** execution.

Intuitively, \( \rightarrow_e \) is an *interleaving* that obeys \( \rightarrow \).
Legal Executions

To see what a parallel program can do, we can enumerate all the SC executions and “run” them:

- $y := 1 \rightarrow \text{tmp1} := x \rightarrow x := 1 \rightarrow \text{tmp2} := y$
Legal Executions

To see what a parallel program can do, we can enumerate all the SC executions and “run” them:

• $y := 1 \rightarrow \text{tmp1} := x \rightarrow x := 1 \rightarrow \text{tmp2} := y$
  $\implies \text{tmp1} \rightarrow 0, \text{tmp2} \rightarrow 1$
Legal Executions

To see what a parallel program can do, we can enumerate all the SC executions and “run” them:

• $y := 1 \quad \rightarrow \quad \text{tmp1} := x \quad \rightarrow \quad x := 1 \quad \rightarrow \quad \text{tmp2} := y$
  $\quad \rightarrow \quad \text{tmp1} \leftrightarrow 0, \text{tmp2} \leftrightarrow 1$

• $y := 1 \quad \rightarrow \quad x := 1 \quad \rightarrow \quad \text{tmp1} := x \quad \rightarrow \quad \text{tmp2} := y$
  $\quad \rightarrow \quad \text{tmp1} \leftrightarrow 1, \text{tmp2} \leftrightarrow 1$
To see what a parallel program can do, we can enumerate all the SC executions and “run” them:

- $y := 1 \rightarrow \text{tmp1} := x \rightarrow x := 1 \rightarrow \text{tmp2} := y$
  $\rightarrow \text{tmp1} \rightarrow 0, \text{tmp2} \rightarrow 1$
- $y := 1 \rightarrow x := 1 \rightarrow \text{tmp1} := x \rightarrow \text{tmp2} := y$
  $\rightarrow \text{tmp1} \rightarrow 1, \text{tmp2} \rightarrow 1$
- $y := 1 \rightarrow x := 1 \rightarrow \text{tmp2} := y \rightarrow \text{tmp1} := x$
  $\rightarrow \text{tmp1} \rightarrow 1, \text{tmp2} \rightarrow 1$
- $x := 1 \rightarrow y := 1 \rightarrow \text{tmp2} := y \rightarrow \text{tmp1} := x$
  $\rightarrow \text{tmp1} \rightarrow 1, \text{tmp2} \rightarrow 1$
- $x := 1 \rightarrow y := 1 \rightarrow \text{tmp1} := x \rightarrow \text{tmp2} := y$
  $\rightarrow \text{tmp1} \rightarrow 1, \text{tmp2} \rightarrow 1$
- $x := 1 \rightarrow \text{tmp2} := y \rightarrow y := 1 \rightarrow \text{tmp1} := x$
  $\rightarrow \text{tmp1} \rightarrow 1, \text{tmp2} \rightarrow 0$
Legal Executions

Enumerating SC executions gets old fast, but lets us produce the set of possible final stores, $\sigma$:

$$\{ \text{tmp1} \mapsto 0, \; \text{tmp2} \mapsto 1 \}$$
$$\{ \text{tmp1} \mapsto 1, \; \text{tmp2} \mapsto 1 \}$$
$$\{ \text{tmp1} \mapsto 1, \; \text{tmp2} \mapsto 0 \}$$

So no sequentially consistent execution makes both tmp1 and tmp2 equal to zero.
volatile int x, y, tmp1, tmp2;

// Thread 0: write x and read y.
void *t0(void *arg) {
    x = 1;
    tmp1 = y;
    return 0;
}

// Thread 1, the opposite: write y and read x.
void *t1(void *arg) {
    y = 1;
    tmp2 = x;
    return 0;
}
void main() {
    x = y = tmp1 = tmp2 = 0;

    // Launch both threads.
    pthread_t thread0, thread1;
    pthread_create(&thread0, NULL, t0, NULL);
    pthread_create(&thread1, NULL, t1, NULL);

    // Wait for both threads to finish.
    pthread_join(thread0, NULL);
    pthread_join(thread1, NULL);

    printf("%d %d\n", tmp1, tmp2);
}
Weak Memory Models

No real parallel machine enforces sequential consistency!
Weak Memory Models

No real parallel machine enforces sequential consistency!

There are many reasons and/or excuses:

- Per-processor caching lets each CPU read values that other processors can’t see yet.
- Private write buffers are critical for good performance with coherent caches.
- Lots of “obvious” compiler optimizations violate sequential consistency.

See also Boehm, 2005: “Threads cannot be implemented as a library.”
Weak Memory Models

Every machine (and every programming language) as a memory model. Memory models describe the set of legal executions.
Weak Memory Models

Every machine (and every programming language) as a memory model. Memory models describe the set of legal executions.

Sequential consistency is the strongest memory model out there: it allows the fewest different executions.

Real machines and languages have weaker memory models:

\[ \text{SC} \geq \text{x86} \geq \text{ARM} \geq \text{C/C++} \geq \text{DRF0} \]
A data race occurs when:

- There are two events $a$ and $b$ that are unordered in the happens-before relation ($a \not\rightarrow b$ and $b \not\rightarrow a$),
- both events access the same shared variable, and
- one or both of $a$ and $b$ is a write.
A data race occurs when:

- There are two events $a$ and $b$ that are unordered in the happens-before relation ($a \not\rightarrow b$ and $b \not\rightarrow a$),
- both events access the same shared variable, and
- one or both of $a$ and $b$ is a write.

Our little example has two data races: one on $x$ and one on $y$. 
Languages have recently agreed on one critical property:

\[
\text{data race free} \implies \text{sequentially consistent}
\]

As long as you avoid data races, you get sequential consistency on any machine in Java, C, and C++.

(In jargon: the DRF implies SC theorem.)
Languages have recently agreed on one critical property:

\[
data \text{ race free } \implies \text{ sequentially consistent}
\]

As long as you avoid data races, you get sequential consistency on \textit{any} machine in Java, C, and C++.

(In jargon: the \textit{DRF implies SC} theorem.)

Languages still disagree about what happens when you \textit{do} have a race. In C and C++, races allow \textit{undefined behavior}. 
Race-Free Programming

Data race detection is an active field of research.

One called **ThreadSanitizer** is included with recent Clang and GCC compilers:

```
$ cc -g -fsanitize=thread simple_race.c
$ ./a.out
WARNING: ThreadSanitizer: data race (pid=26327)
  Write of size 4 at 0x7f89554701d0 by thread T1:
    #0 Thread1(void*) simple_race.cc:8

  Previous write of size 4 at 0x7f89554701d0 by thread T2:
    #0 Thread2(void*) simple_race.cc:13
```