Concurrency & Parallelism

Nick Sumner
wsumner@sfu.ca
Concurrency & Parallelism
Seeking out performance

- Improving performance can come from tuning
  - Algorithmic complexity
  - Memory access patterns
  - Concurrency
  - Parallelism
Seeking out performance

- Improving performance can come from tuning
  - Algorithmic complexity
  - Memory access patterns
  - Concurrency
  - Parallelism

- As processor speeds have slowed increasing, much focus has been placed on the last two
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
Concurrent parsing

Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.

- **Parallelism** is using multiple resources to perform multiple tasks at the same time.
  - e.g. multiple cores for tasks, vector instructions
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.

- **Parallelism** is using multiple resources to perform multiple tasks at the same time.
  - e.g. multiple cores for tasks, vector instructions
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined.
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined.

- The best possible running time is determined by the critical path or span of dependent tasks through the program.
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined.

- The best possible running time is determined by the *critical path* or *span* of dependent tasks through the program.
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined. This is too optimistic! Why?

- The best possible running time is determined by the critical path or span of dependent tasks through the program.
Using Parallelism

- There are often more tasks than compute resources
Using Parallelism

- There are often more tasks than compute resources
  - Brent’s Theorem describes the time accounting for limits

\[
\text{Given } p \text{ processors, } \frac{\text{Time}_1}{p} \leq \text{Time}_p \leq \frac{\text{Time}_1}{p} + \text{Time}_\infty
\]
Using Parallelism

- There are often more tasks than compute resources
  - Brent’s Theorem describes the time accounting for limits

\[
\text{Given } p \text{ processors, } \frac{\text{Time}_1}{p} \leq \text{Time}_p \leq \frac{\text{Time}_1}{p} + \text{Time}_\infty
\]

- Identifying good opportunities for effective parallelism is open to research
Using Parallelism

- There are often more tasks than compute resources
  - Brent’s Theorem describes the time accounting for limits

\[
\text{Given } p \text{ processors, } \frac{\text{Time}_1}{p} \leq \text{Time}_p \leq \frac{\text{Time}_1}{p} + \text{Time}_\infty
\]

- Identifying good opportunities for effective parallelism is open to research
  - Profiling for tasks to extract
  - Understanding the effect of speeding specific tasks
  - ...
Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others
Correctness issues

• Parallel & concurrent code is challenging to write
  – Nondeterministic timing
  – Actions of one task may subtly affect others

• Specifically
  – Deadlock / Livelock
  – Starvation
  – Data races
  – Atomicity violations
  – Order violations
  – ...
Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others

- Specifically
  - Deadlock / Livelock
  - Starvation
  - Data races
  - Atomicity violations
  - Order violations
  - ...

97% of real world concurrency bugs

[Lu, ASPLOS 2008]
Data Races

- A data race occurs when:
Data Races

- A data race occurs when:
  1) two threads access the same location
A data race occurs when:

1) two threads access the same location
2) the accesses are *logically simultaneous*
A data race occurs when:
1) two threads access the same location
2) the accesses are *logically simultaneous*
3) at least one access is a write (WAW, WAR, RAW)
Data Races

- A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*
  3) at least one access is a write (WAW, WAR, RAW)
Data Races

T1  T2

x++

tmp = x
tmp = tmp + 1
x = tmp

tmp1 = x
tmp1 = tmp1 + 1
x = tmp1

tmp2 = x
tmp2 = tmp2 + 1
x = tmp2
Data Races

```plaintext
x++

T1  T2

tmp = x
tmp = tmp+1
x = tmp

tmp1 = x
tmp1 = tmp1+1
x = tmp1

x \leftrightarrow 0

tmp2 = x
tmp2 = tmp2+1
x = tmp2
```
Data Races

x++

T1

T2

\[
\begin{align*}
\text{tmp}_1 &= x \\
\text{tmp}_1 &= \text{tmp}_1 + 1 \\
x &= \text{tmp}_1
\end{align*}
\]

\[
\begin{align*}
\text{tmp}_2 &= x \\
\text{tmp}_2 &= \text{tmp}_2 + 1 \\
x &= \text{tmp}_2
\end{align*}
\]
Data Races

x++

\[
\begin{align*}
\text{tmp} &= x \\
\text{tmp} &= \text{tmp} + 1 \\
x &= \text{tmp}
\end{align*}
\]

T1  T2

\[
\begin{align*}
\text{tmp}_1 &= x \\
\text{tmp}_1 &= \text{tmp}_1 + 1 \\
x &= \text{tmp}_1
\end{align*}
\]

\[
\begin{align*}
\text{tmp}_2 &= x \\
\text{tmp}_2 &= \text{tmp}_2 + 1 \\
x &= \text{tmp}_2
\end{align*}
\]
Data Races

T1  T2

x++

tmp = x
tmp = tmp + 1
x = tmp

T1  T2

tmp1 = x
tmp1 = tmp1 + 1
x = tmp1

tmp2 = x
tmp2 = tmp2 + 1
x = tmp2

tmp1 = x
tmp1 = tmp1 + 1
x = tmp1

tmp2 = x
tmp2 = tmp2 + 1
x = tmp2

x = tmp1

x = tmp2
Data Races

T1 T2

x++

tmp = x
tmp = tmp+1
x = tmp

T1 T2

tmp1 = x
tmp1 = tmp1+1
x = tmp1

tmp2 = x
tmp2 = tmp2+1
x = tmp2

tmp1 = x
tmp1 = tmp1+1
x = tmp1

tmp2 = x
tmp2 = tmp2+1
x = tmp2
Data Races

x++

T1

tmp = x

T2
tmp = tmp + 1

x = tmp

tmp1 = x
tmp1 = tmp1 + 1
x = tmp1

tmp2 = x
tmp2 = tmp2 + 1
x = tmp2

tmp1 = x
tmp1 = tmp1 + 1

T1

x ⇔ 0

T2

tmp1 ⇔ 0

tmp2 = x
tmp2 = tmp2 + 1
x = tmp2

x 0 → tmp1 0 →
Data Races

```
x++

tmp = x
tmp = tmp + 1
x = tmp
```

T1 T2

```
tmp1 = x
tmp1 = tmp1 + 1
x = tmp1
```

```
tmp2 = x
tmp2 = tmp2 + 1
x = tmp2
```

T1 T2

```
x ⇔ 0
```

```
tmp1 ⇔ 0
```

```
tmp2 = x
tmp2 ⇔ 0
```

```
tmp2 = tmp2 + 1
x = tmp2
```
Data Races

x++

tmp = x
tmp = tmp + 1
x = tmp

tmp₁ = x
tmp₁ = tmp₁ + 1
x = tmp₁

tmp₂ = x
tmp₂ = tmp₂ + 1
x = tmp₂

x 0 ↦

x 1 ↦
tmp₁ 0 ↦
tmp₂ 0 ↦
tmp₁ 1 ↦
tmp₂ 1 ↦

T1   T2
  T1   T2
  T1   T2
  T1   T2
Data Races

x++

\[
\begin{align*}
\text{tmp} &= x \\
\text{tmp} &= \text{tmp} + 1 \\
\text{x} &= \text{tmp}
\end{align*}
\]

T1  T2

T1  T2

\[
\begin{align*}
\text{tmp}_1 &= x \\
\text{tmp}_1 &= \text{tmp}_1 + 1 \\
\text{x} &= \text{tmp}_1
\end{align*}
\]

\[
\begin{align*}
\text{tmp}_2 &= x \\
\text{tmp}_2 &= \text{tmp}_2 + 1 \\
\text{x} &= \text{tmp}_2
\end{align*}
\]

x 0 ↦ x 1 ↦ x 1 ↦

tmp 0 ↦ tmp 0 ↦

x 0 ↦ tmp 1 0 ↦

tmp 2 0 ↦ tmp 2 0 ↦

x 1 ↦ tmp 1 1 ↦

tmp 2 1 ↦ tmp 2 1 ↦
Data Races

Synchronization discipline prevents data races.
Data Races

$\texttt{x++}$

T1

$\texttt{tmp = x}$
$\texttt{tmp = tmp + 1}$
$\texttt{x = tmp}$

T2

lock(m)
$\texttt{x++}$
unlock(m)

T1

T2

$\texttt{tmp_1 = x}$
$\texttt{tmp_1 = tmp_1 + 1}$
$\texttt{x = tmp_1}$

$\texttt{tmp_2 = x}$
$\texttt{tmp_2 = tmp_2 + 1}$
$\texttt{x = tmp_2}$

$\texttt{tmp_1 = x}$
$\texttt{tmp_1 = tmp_1 + 1}$

$\texttt{tmp_2 = x}$
$\texttt{tmp_2 = tmp_2 + 1}$
$\texttt{x = tmp_2}$
Data Races

T1  T2

x++

tmp = x
tmp = tmp + 1
x = tmp

lock(m)
x++
unlock(m)

T1  T2

tmp1 = x
tmp1 = tmp1 + 1
x = tmp1

lock(m)
tmp1 = x

tmp1 = tmp1 + 1

T1  T2

tmp2 = x
tmp2 = tmp2 + 1
x = tmp2
Data Races

T1  T2

\[
\begin{align*}
\text{tmp} &= x \\
\text{tmp} &= \text{tmp} + 1 \\
x &= \text{tmp}
\end{align*}
\]

\[
\begin{align*}
\text{lock}(m) \\
x++ \\
\text{unlock}(m)
\end{align*}
\]

T1  T2

\[
\begin{align*}
\text{tmp}_1 &= x \\
\text{tmp}_1 &= \text{tmp}_1 + 1 \\
x &= \text{tmp}_1
\end{align*}
\]

T1  T2

\[
\begin{align*}
\text{lock}(m) \\
\text{tmp}_1 &= x \\
\text{tmp}_1 &= \text{tmp}_1 + 1
\end{align*}
\]

The second task must wait.
Data Races

T1  T2

lock(m)
++
unlock(m)

T1  T2

lock(m)
++
unlock(m)

x++
tmp = x
tmp = tmp+1
x = tmp

T1  T2

lock(m)
tmp1 = x
tmp1 = tmp1+1
x = tmp1

T1  T2

lock(m)
tmp1 = x
tmp1 = tmp1+1
x = tmp1

T1  T2

lock(m)
tmp2 = x
tmp2 = tmp2+1
x = tmp2

T1  T2

lock(m)
tmp2 = x
tmp2 = tmp2+1
x = tmp2
Data Races

T1       T2

x++

T1       T2

lock(m)

x++

unlock(m)

T1       T2

lock(m)

tmp = x
tmp = tmp + 1
x = tmp

T1       T2

lock(m)

tmp = x
tmp = tmp + 1
x = tmp

T1       T2

lock(m)

tmp = x
tmp = tmp + 1
x = tmp

T1       T2

lock(m)

tmp = x
tmp = tmp + 1
x = tmp

T1       T2

lock(m)

tmp = x
tmp = tmp + 1
x = tmp
“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless
“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless

- Both programming languages and hardware have memory models that determine what is really okay
"Benign" Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless

- Both programming languages and hardware have memory models that determine what is really okay
  - A memory model determines what values may be read by a given memory access, esp. w.r.t. previous writes

[CACM 2010, PLDI 2018]
“Benign” Data Races

Sometimes developers want to lazily initialize data even with multiple threads.

if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;

[Boehm, Hotpar 2011]
“Benign” Data Races

- Threads race on `init`
- The compiler assumes no races while optimizing

```c
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;
```

[Boehm, Hotpar 2011]
“Benign” Data Races

- Threads race on `init`
- The compiler assumes no races while optimizing

```c
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;
```

[Boehm, Hotpar 2011]
“Benign” Data Races

- Threads race on `init`
- The compiler assumes no races while optimizing

```
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}

if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}

Data can even be loaded early. This can even be done just by hardware!
```

[Boehm, Hotpar 2011]
Sometimes developers want to interleave conditional behavior.

```
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}
```

[Boehm, Hotpar 2011]
“Benign” Data Races

- Data race freedom allows extra reads.

```java
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}
```

[Boehm, Hotpar 2011]
“Benign” Data Races

- Data race freedom allows extra reads.

```plaintext
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}
```

[Boehm, Hotpar 2011]
“Benign” Data Races

- Data race freedom allows extra reads.

local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}

local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (counter > localMax) {
    handler(update);
}

[Boehm, Hotpar 2011]

Other Thread

counter = ...;

False

True
“Benign” Data Races

- Races can introduce bugs on non-racy variables

\[
\begin{align*}
c &= a + 10 \\
\ldots \\
b &= a + 10 \\
c &= 1
\end{align*}
\]

[Dolan, PLDI 2018]
“Benign” Data Races

- Races can introduce bugs on non-racy variables

\[ c = a + 10 \]
\[
\ldots
\]
\[ b = a + 10 \]

[Dolan, PLDI 2018]

\[ c = 1 \]

\[ c = a + 10 \]
\[
\ldots
\]
\[ b = c \]

rematerialization

\[ c = 1 \]
“Benign” Data Races

- Races can jump forward and backward in time

```
a = 1
flag = true

T1
```

```
a = 2
f = flag
b = a
c = a

T2
```

[Dolan, PLDI 2018]
“Benign” Data Races

- Races can jump forward and backward in time

This can happen in Java when `flag` is volatile & `b` is a complex reference

[Dolan, PLDI 2018]
“Benign” Data Races

- Races can jump forward and backward in time

This can happen in Java when flag is volatile & b is a complex reference

T1 T2

\[
\begin{align*}
a &= 1 \\
\text{flag} &= \text{true} \\
a &= 2 \\
f &= \text{flag} \\
b &= a \\
c &= a
\end{align*}
\]

[Dolan, PLDI 2018]

T1 T2

\[
\begin{align*}
a &= 1 \\
\text{flag} &= \text{true} \\
a &= 2 \\
f &= \text{flag} \\
b &= a \\
c &= 2
\end{align*}
\]

2 can be read after 1 even in the same thread!
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*)
  - defined behavior occurs when writes & reads are ordered
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps *per individual variable* order
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps *per individual variable* order
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: \textit{simultaneously})
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, \textcolor{red}{fork/join} constrain order
  - access to volatile variables keeps \textit{per individual variable} order

lock()
...
unlock()

lock()
...
unlock()
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps per individual variable order

- Happens-Before ordering of a specific execution can be tracked to identify bugs
Happens-Before Ordering

T1  T2

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
Happens-Before Ordering

T1  T2

lock()
tmp = x
tmp = tmp + 1
x = tmp
unlock()

lock()
tmp = x
tmp = tmp + 1
x = tmp
unlock()
Happens-Before Ordering

T1  T2

lock()
tmp = x
tmp = tmp + 1
x = tmp
unlock()
Happens-Before Ordering

T1  T2

T1
lock()
tmp = x
tmp = tmp + 1
x = tmp
unlock()

T2
lock()
tmp = x
tmp = tmp + 1
x = tmp
unlock()
Happens-Before Ordering

T1  T2

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()

T1  T2

tmp = x
tmp = tmp+1
x = tmp

No ordering!
Simultaneous!
Race detected!

tmp = x
tmp = tmp+1
x = tmp
Happens-Before Ordering

- Note, this only detects races in the current execution & schedule! Why?
Happens-Before Ordering

- Note, this only detects races in the current execution & schedule!

Why?

T1

\[
y = \ldots;
\]

lock()

x++

unlock()

T2

lock()

x++

unlock()

\[
\ldots = y;
\]
Happens-Before Ordering

- Note, this only detects races in the current execution & schedule!

Why?

T1

y = ...;
lock()
x++
unlock()

lock()
x++
unlock()

... = y;

T2

lock()
x++
unlock()

... = y;

Why?
Happens-Before Ordering

- Note, this only detects races in the current execution & schedule!
  - *Sound predictive* data race detection can extend it *across other executions* [PLDI 2017/2018, 2020]
Happens-Before Ordering

- Note, this only detects races in the current execution & schedule!
  - *Sound predictive* data race detection can extend it across other executions [PLDI 2017/2018, 2020]

- Requires careful tracking of dependences
  - Careful construction of logical time using *vector clocks* [JVM 2001, PLDI 2009]
What is the happens-before relation?

T1

lock(a)
write x
unlock(a)

write x

T2

write y

lock(a)
read x
unlock(a)
Logical Time & Vector Clocks

What is the happens-before relation?

T1

lock(a)
write x
unlock(a)
write x

T2

write y
lock(a)
read x
unlock(a)
What is the happens-before relation?

Logical Time & Vector Clocks

T1  T2

- lock(a)
- write x
- unlock(a)
- write x
- unlock(a)
- write y
- lock(a)
- read x
- unlock(a)
Logical Time & Vector Clocks

What is the happens-before relation?

T1

lock(a)
write x
unlock(a)
write x

T2
write y
lock(a)
read x
unlock(a)
What is the happens-before relation?

Logical Time & Vector Clocks
Logical Time & Vector Clocks

What is the happens-before relation?

lock(a)
write x
unlock(a)
write x
unlock(a)
lock(a)
read x
unlock(a)
write y
T1 T2
Logical Time & Vector Clocks

T1  T2

lock(a)  write x  unlock(a)
write x
lock(a)  read x  unlock(a)

write y

Clocks

T1↦[0,0]
T2↦[0,0]

Shadow Memory

x↦R[0,0]
W[0,0]
y↦R[0,0]
W[0,0]
Logical Time & Vector Clocks

lock(a)
write x
unlock(a)
write x
lock(a)
read x
unlock(a)

write y

Clocks
T1⇒[0,0]
T2⇒[0,0]

Shadow Memory
x⇒R[0,0]
W[0,0]
y⇒R[0,0]
W[0,0]
Logical Time & Vector Clocks

T1  T2

lock(a)  write y  lock(a)
write x   read x   write x
unlock(a)  unlock(a)  y ↦ R[0,0]

T1 ↦ [0,0]
T2 ↦ [0,0]

x ↦ R[0,0]  W[0,0]
y ↦ R[0,0]  W[0,0]

Shadow Memory
Logical Time & Vector Clocks

\[ x \mapsto \mathbb{R}_{[0,0]} \]
\[ y \mapsto \mathbb{R}_{[0,0]} \]

\( \text{lock}(a) \)
\( \text{write } x \)
\( \text{lock}(a) \)
\( \text{read } x \)
\( \text{unlock}(a) \)
\( \text{unlock}(a) \)
\( \text{write } x \)

Shadow Memory

\( T_1 \mapsto [1,0] \)
\( T_2 \mapsto [0,0] \)

\[ x \mapsto \mathbb{R}_{[0,0]} \]
\[ y \mapsto \mathbb{R}_{[0,0]} \]
\[ W_{[0,0]} \]
Logical Time & Vector Clocks

Clocks
- T1 $\mapsto [1,0]
- T2 $\mapsto [0,0]

Shadow Memory
- x $\mapsto R[0,0]
- y $\mapsto R[0,0]
- x $\mapsto W[1,0]
- y $\mapsto W[0,0]

Timeline:
- Lock(a)
- Write x
- Unlock(a)
- Write y
- Lock(a)
- Read x
- Unlock(a)
Logical Time & Vector Clocks

Clocks

T1 \mapsto [2,0]
T2 \mapsto [0,0]

Shadow Memory

x \mapsto R[0,0]
y \mapsto R[0,0]
a \mapsto [1,0]
W[1,0]
W[0,0]
Logical Time & Vector Clocks

[Diagram showing logical time and vector clocks with examples of lock, write, unlock, and sendProgress operations.]
Logical Time & Vector Clocks

Clocks

T1 \rightarrow [2, 0]
T2 \rightarrow [1, 1]

Shadow Memory

x \rightarrow R[0, 0]
W[1, 0]
y \rightarrow R[0, 0]
W[0, 0]
a \rightarrow [1, 0]
Logical Time & Vector Clocks

Clocks

T1 \rightarrow [2, 0]
T2 \rightarrow [1, 1]

Shadow Memory

a \rightarrow [1, 0]
x \rightarrow R[0, 0]
W[1, 0]
y \rightarrow R[0, 0]
W[0, 0]

receiveProgress(T2, a)
advanceLocal(T2)

lock(a) write x unlock(a)
lock(a) read x unlock(a)

write x

write y
Logical Time & Vector Clocks

Clocks

T1 → [2, 0]
T2 → [1, 1]

Shadow Memory

x → R[1, 1]
y → R[0, 0]
W[1, 0]
W[0, 0]

lock(a)
write x
unlock(a)

lock(a)
write y
unlock(a)

write x
read x
unlock(a)
Logical Time & Vector Clocks

**Clocks**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2, 0]</td>
<td>[1, 2]</td>
</tr>
</tbody>
</table>

**Shadow Memory**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>R[1, 1]</td>
</tr>
<tr>
<td>w</td>
<td>W[1, 0]</td>
</tr>
<tr>
<td>y</td>
<td>R[0, 0]</td>
</tr>
<tr>
<td>w</td>
<td>W[0, 0]</td>
</tr>
</tbody>
</table>
Logical Time & Vector Clocks

Clocks

\[ T1 \mapsto [2, 0] \]
\[ T2 \mapsto [1, 2] \]

Shadow Memory

\[ x \mapsto R[1, 1] \]
\[ y \mapsto R[0, 0] \]
\[ W[2, 0] \]
\[ W[0, 0] \]
Logical Time & Vector Clocks

Clocks

\[ T_1 \mapsto [2, 0] \]
\[ T_2 \mapsto [1, 2] \]

Shadow Memory

\[ x \mapsto R[1, 1] \]
\[ y \mapsto R[0, 0] \]
\[ W[2, 0] \]
\[ W[0, 0] \]

\[ C(R, X) \nleq C(W, X) \]
\[ & C(W, X) \nleq C(R, X) \rightarrow \text{race!} \]
Logical Time & Vector Clocks

**Clocks**
- \( T_1 \mapsto [2, 0] \)
- \( T_2 \mapsto [1, 2] \)

**Shadow Memory**
- \( x \mapsto R[1, 1] \)
- \( y \mapsto R[0, 0] \)
- \( W[2, 0] \)
- \( W[0, 0] \)

When reading:
- check against the last write

When writing:
- check against the last read and write

**Formula**
- \( C(R, X) \not\subset C(W, X) \)
- \( C(W, X) \not\subset C(R, X) \)

\( \rightarrow \text{race!} \)
Vector Clocks

- Vector clocks have a long history in reasoning about distributed systems and concurrency
Vector Clocks

- Vector clocks have a long history in reasoning about distributed systems and concurrency

- With many threads/parallel tasks, clock overheads increase
  - With effort, newer approaches optimize and replace vector clocks [Mathur 2021]
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!

```
T1
lock(a); lock(b)
read y
write x
unlock(b)
write x
unlock(a)

T2
lock(b)
write x
unlock(b)
unlock(a)
```
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!

```
lock(a); lock(b)
read y
write x
unlock(b)
write x
unlock(a)
```

```
lock(b)
write x
unlock(b)
```

Note: Both \(x\) and \(y\) are always protected by locks. \(x\) still races.
Data Race Detection - Locksets

- Lockset based data race detection has many issues
  - Synchronization may be fork/join, wait/notify based
  - Initialization --> Process in Parallel --> Combine
  - Richer parallel designs
Data Race Detection - Locksets

- Lockset based data race detection has many issues
  - Synchronization may be fork/join, wait/notify based
  - Initialization --> Process in Parallel --> Combine
  - Richer parallel designs

- Tends to have many false positives
Data Race Detection - Locksets

- Lockset based data race detection has many issues
  - Synchronization may be fork/join, wait/notify based
  - Initialization --> Process in Parallel --> Combine
  - Richer parallel designs

- Tends to have many false positives

Can you think of concurrency bugs it will miss, too?
Order Violations

- Some accesses are wrongly assumed to occur before others

```
x = new Data
x->datum
```

wait/notify or condition variables can fix these
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

```c
| tmp = x
| tmp = tmp+1
| x = tmp
```
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

```
lock()
tmp = x
unlock()

lock()
tmp = tmp+1
unlock()
x = tmp
```

**No race, similar effect!**
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

```plaintext
What do we really want?

```

```plaintext
lock()
tmp = x
unlock()

lock()
tmp = tmp+1
unlock()

lock()
x = tmp
unlock()
```

```plaintext
VS

tmp = x
tmp = tmp+1
x = tmp
```
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is atomic if it is equivalent to a sequentially executed one.
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is \textit{atomic} if it is equivalent to a sequentially executed one.
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is **atomic** if it is equivalent to a sequentially executed one.

```
Acq(m)  Rd(x,0)  Wr(x,1)  Rel(m)
```

```
Acq(m)  Rd(x,0)  Wr(x,1)  Rel(m)
```
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is *atomic* if it is equivalent to a sequentially executed one.
  - This also takes care of data races
  - Similar to notions from databases (serializability & linearizability)
How can we find atomicity violations (or correctness)?
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]

```
Acq(m) ─ Rd(x,0) ─ Wr(x,1) ─ Rel(m)
```
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]

Acq(m) — Rd(x,0) — Wr(x,1) — Rel(m)

Acq(m) — Rd(x,0) — Wr(x,1) — Rel(m)
How can we find atomicity violations (or correctness)?

- Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
- 2 thread atomicity patterns [Lu ASPLOS ‘06]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]

Only some patterns are unserializable. Detect unlikely issues via training.
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]

\[
\begin{align*}
a &= x \\
\text{e} &= \text{d} \\
\text{c} &= x \\
\text{d} &= \text{c} \\
\end{align*}
\]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  – Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  – 2 thread atomicity patterns [Lu ASPLOS ‘06]
  – Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ’75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]

\[
\begin{align*}
\text{a} &= \text{x} \\
\text{c} &= \text{x} \\
\text{d} &= \text{c} \\
\text{e} &= \text{d}
\end{align*}
\]

Cycles are unserializable!
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]

- How do we know what regions should be atomic?
Concurrent in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs

```
<html><body>
<button id="b1" onclick="fn()">b1</button>
<script>
  function fn(){
    m=null;
  }
</script>
<script src="lib.js"></script>
<script>
  m={data:"
</script>
<script src="extn.js"></script>
<script>
  m.data ="text"
</script>
</body></html>
```
Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs

```html
<html><body>
<button id="b1" onclick="fn()">b1</button>

<script>
  function fn(){
    m=null;
  }
</script>

<script src="lib.js"></script>
<script>
  m={data:"
</script>

<script src="extn.js"></script>
<script>
  m.data ="text"
</script>
</body></html>
```
Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs

[ Hong 2014 ]

```html
<html><body>
<button id="b1" onclick="fn()">b1</button>
<script>
function fn(){
m=null;
}
</script>
<script src="lib.js"></script>
<script>
m={data:""};
</script>
<script src="extn.js"></script>
</body></html>

A user can click “here”

Error: m.data is undefined (line 13)
Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs
  - While the meaning may be well defined, it can be unexpected & confusing.
Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs
  - While the meaning may be well defined, it can be unexpected & confusing.

- In practice, semantic concurrency bugs exist in Javascript!
Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs
  - While the meaning may be well defined, it can be unexpected & confusing.

- In practice, semantic concurrency bugs exist in Javascript!
  - Both in client side & server side [Raychev 2013, Hong 2014, Endo 2020]
Concurrence in the land of Single Threads

- Even programs without threads face concurrency bugs

Does your program have events?
- Can one event lead to another event?
- The order of events can lead to bugs
- While the meaning may be well defined, it can be unexpected & confusing.

In practice, semantic concurrency bugs exist in Javascript!

```
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
    entries++;
    fs.lstat(filePath, function stat(err, stats) {
        if (err) {
            entries--;
            return;
        }
        useStatData(stats);
        fs.readFile(filePath, function read(err, data) {
            performTask(data);
            doneTasks++;
            if (doneTasks === entries)
                finalize();
        });
    });
}
```

[Endo 2020]
Even programs without threads face concurrency bugs. Does your program have events? Can one event lead to another event? The order of events can lead to bugs. While the meaning may be well defined, it can be unexpected & confusing.

In practice, semantic concurrency bugs exist in Javascript! [Endo 2020, Raychev 2013, Hong 2014, Endo 2020]

```javascript
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
    entries++;
    fs.lstat(filePath, function stat(err, stats) {
        if (err) {
            entries--; return;
        }
        useStatData(stats);
        fs.readFile(filePath, function read(err, data) {
            performTask(data);
            doneTasks++;
            if (doneTasks === entries)
                finalize();
        });
    });
}
```
Concurrence in the land of Single Threads

- Even programs without threads face concurrency bugs

```javascript
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
    entries++;
    fs.lstat(filePath, function stat(err, stats) {
        if (err) {
            entries--;
            return;
        }
        useStatData(stats);
        fs.readFile(filePath, function read(err, data) {
            performTask(data);
            doneTasks++;
            if (doneTasks === entries)
                finalize();
        });
    });
}
```

[Endo 2020]
Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs

```javascript
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
  entries++;
  fs.lstat(filePath, function stat(err, stats) {
    if (err) {
      entries--;
      return;
    }
    useStatData(stats);
    fs.readFile(filePath, function read(err, data) {
      performTask(data);
      doneTasks++;  // Update doneTasks when readFile completes
      if (doneTasks === entries)
        finalize();
    });
  });
}

processFile('existing-file.txt');
processFile('missing-file.txt');
processFile('empty-file.txt');
```

[Endo 2020]
Concurrence in the land of Single Threads

- Even programs without threads face concurrency bugs

Even programs without threads face concurrency bugs – Does your program have events – Can one event lead to another event – The order of events can lead to bugs – While the meaning may be well defined, it can be unexpected & confusing.

In practice, semantic concurrency bugs exist in Javascript! – [Raychev 2013, Hong 2014, Endo 2020]

```javascript
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
  entries++;
  fs.lstat(filePath, function stat(err, stats) {
    if (err) {
      entries--;
      return;
    }
    useStatData(stats);
    fs.readFile(filePath, function read(err, data) {
      performTask(data);
      doneTasks++;
      if (doneTasks === entries)
        finalize();
    });
  });
}

processFile('existing-file.txt');
processFile('missing-file.txt');
processFile('empty-file.txt');
```

When the missing file stats last, finalize is not called, and Node hangs.
Concurrent in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs
  - While the meaning may be well defined, it can be unexpected & confusing.

- In practice, semantic concurrency bugs exist in Javascript!
  - Both in client side & server side [Raychev 2013, Hong 2014, Endo 2020]
  - In fact, locks are simulated to control interleavings

- Careful (1) monitoring, (2) happens-before analysis, & (3) test generation can automatically find these issues
Concurrent Test Generation

- What if we don’t already have a buggy execution?
Concurrent Test Generation

- What if we don’t already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs
Concurrent Test Generation

- What if we don’t already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs
- Careful schedule generation & selection
Concurrent Test Generation

- What if we don’t already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs
- Careful schedule generation & selection
- Generate API unit tests targeting concurrency
  - Small enough for exhaustive schedule exploration
Other Directions

- Shepherding toward good behaviors
- Tolerating & avoiding on the fly
- Static analysis
Summary

- Parallelism is important for modern performance
- Choosing what to parallelize can be hard
- Parallelizing correctly can be very hard
Summary

- Parallelism is important for modern performance
- Choosing what to parallelize can be hard
- Parallelizing correctly can be very hard

And the hard problems are interesting to study.