Concurrency & Parallelism
CSE 6341

Readings:
Memory Models (CACM ’10),
Java Memory Model (POPL ’05) (Sec. 1-4),
Case for Concurrency Exceptions (HotPar ’09)
What are the possible behaviors?

Initially:
```
int data = 0;
boolean flag = false;
```

**T1:**
```
data = ...;
flag = true;
```

**T2:**
```
if (flag)
  ... = data;
```
Hardware becoming more parallel
*more* instead of *faster* cores

Software must become more parallel
“From my perspective, parallelism is the biggest challenge since high-level programming languages. It’s the biggest thing in 50 years because industry is betting its future that parallel programming will be useful.

“Industry is building parallel hardware, assuming people can use it. And I think there's a chance they'll fail since the software is not necessarily in place. So this is a gigantic challenge facing the computer science community. If we miss this opportunity, it's going to be bad for the industry.”

—David Patterson, ACM Queue interview, 2006
Shared memory programming

- Imperative programs
  - Java, C#, C, C++, Python, Ruby
- Threads
  - Shared, mutable state
  - Synchronization primitives:
    - Lock acquire & release
    - Monitor wait & notify
    - Thread start & join
Concurrency bugs: atomicity, order, & sequential consistency violations

Concurrent & correct

Concurrency bug: deadlock

Poor performance: lock contention, serialization
DRFo [Adve & Hill ’90]

- C++11 memory model [Boehm & Adve ‘08]
- Java memory model [Manson et al. ‘05]
Program executions have different behaviors

Sequentially consistent

Region serializability of synchronization-free regions
DRF0: strong guarantees for race-free programs

Sequentially consistent

Region serializability of synchronization-free regions
Region serializability of synchronization-free regions

Sequentially consistent

DRF0: no guarantees for racy programs
DRF0: no guarantees for racy programs

Sequentially consistent

Region serializability of synchronization-free regions

Data races are here to stay
Dynamic data race detection [Flanagan & Freund ‘09]
Static data race detection [Naik & Aiken ‘07]
Data-race-free language design [Boyapati et al. ‘02]
DRF0 [Adve & Hill ’90]

C++11 memory model [Boehm & Adve ’08]
- Data races → no semantics

Java memory model [Manson et al. ’05]
- Data races → only type & memory safety preserved
- But... the JMM is actually broken
Data races

- Two accesses to same variable
- At least one is a write

Not well-synchronized
(not ordered by *happens-before* relationship)

Or: accesses can happen simultaneously
Is there a data race?

Initially:

```java
int data = 0;
boolean flag = false;
```

**T1:**

```java
data = ...;
flag = true;
```

**T2:**

```java
if (flag)
...
    = data;
```
Is there a data race?

Initially:

```java
int data = 0;
boolean flag = false;
```

**T1:**

data = ...;

```java
flag = true;
```

**T2:**

```java
if (flag)
    ...
    = data;
```
Is there a data race?

Initially:

```java
int data = 0;
boolean flag = false;
```

T1:

```java
data = ...;
flag = true;
```

T2:

```java
if (flag)
... = data;
```
Possible behavior

Initially:
int data = 0;
boolean flag = false;

T1:
flag = true;

data = ...;

T2:
if (flag)
    ... = data;

Possible behavior

Initially:
int data = 0;
boolean flag = false;

T1:

data = ...;
flag = true;

T2:

tmp = data;

if (flag)
... = tmp;
Is there a data race?

Initially:
```java
int data = 0;
boolean flag = false;
```

**T1:**
```java
data = ...;
synchronized (m) {
    flag = true;
}
```

**T2:**
```java
boolean tmp;
synchronized (m) {
    tmp = flag;
}
if (tmp)
    ... = data;
```
Is there a data race?

Initially:
int data = 0;
boolean flag = false;

T1:
data = ...;
acquire(m);
    flag = true;
release(m);

T2:
boolean tmp;
acquire(m);
tmp = flag;
release(m);
if (tmp)
    ... = data;
Is there a data race?

Initially:
int data = 0;
boolean flag = false;

T1:
data = ...;
acquire(m);
    flag = true;
release(m);

T2:
boolean tmp;
acquire(m);
    tmp = flag;
release(m);
if (tmp)
    ... = data;

Compiler and hardware obey rules about reordering across synchronization
Is there a data race?

Initially:

```java
int data = 0;
volatile boolean flag = false;
```

**T1:**

```java
data = ...;
flag = true;
```

**T2:**

```java
if (flag)
    ... = data;
```

Happens-before relationship
What can this program print?

Initially:
\begin{verbatim}
int data = 0;
boolean flag = false;
\end{verbatim}

**T1:**
\begin{verbatim}
data = 42;
flag = true;
\end{verbatim}

**T2:**
\begin{verbatim}
while (!flag) {
}
print(data);
\end{verbatim}
What can this program print?

Initially:

```java
int data = 0;
boolean flag = false;
```

**T1:**

```java
data = 42;
flag = true;
```

**T2:**

```java
boolean f;
do {
    f = flag;
} while (!f);
int d = data;
print(d);
```
What can this program print?

Initially:

```java
int data = 0;
boolean flag = false;
```

**T1:**

flag = true;

data = 42;

**T2:**

```java
boolean f;
do {
    f = flag;
} while (!f);
int d = data;
print(d);
```
What can this program print?

Initially:
int data = 0;
boolean flag = false;

**T1:**
data = 42;
flag = true;

**T2:**
int d = data;
boolean f;
do {
    f = flag;
} while (!f);
print(d);
Another example: double-checked locking

class Movie {
    Vector<String> comments;

    addComment(String s) {
        if (comments == null) {
            comments = new Vector<String>();
        }
        comments.add(s);
    }
}
class Movie {
    Vector<String> comments;

    addComment(String s) {
        synchronized (this) {
            if (comments == null) {
                comments = new Vector<String>();
            }
            comments.add(s);
        }
    }
}
Another example: double-checked locking

class Movie {
    Vector<String> comments;

    addComment(String s) {
        if (comments == null) {
            synchronized (this) {
                if (comments == null) {
                    comments = new Vector<String>();
                }
            }
        }
        comments.add(s);
    }
}
Another example: double-checked locking

```java
addComment(String s) {
    synchronized (this) {
        if (comments == null) {
            comments = new Vector<String>();
        }
    }
    comments.add(s);
}
```

```java
addComment(String s) {
    if (comments == null) {
        comments = new Vector<String>();
    }
    comments.add(s);
}
```
Another example: double-checked locking

```java
addComment(String s) {
    if (comments == null) {
        synchronized (this) {
            if (comments == null) {
                Vector temp = alloc Vector;
                temp.<init>();
                comments = temp;
            }
        }
        comments.add(s);
    } else {
        comments.add(s);
    }
}
```
Another example: double-checked locking

```java
addComment(String s) {
    if (comments == null) {
        synchronized (this) {
            if (comments == null) {
                Vector temp = alloc Vector;
                temp.<init>();
                comments = temp;
            }
        }
    }
    comments.add(s);
}
```

```java
addComment(String s) {
    if (comments == null) {
        Vector temp = alloc Vector;
        temp.<init>();
        comments = temp;
    }
    comments.add(s);
}
```
Another example: double-checked locking

```java
addComment(String s) {
    if (comments == null) {
        synchronized (this) {
            if (comments == null) {
                Vector temp = alloc Vector;
                comments = temp;
                temp.<init>();
            }
            comments.add(s);
        }
    }
}
```
DRF0 [Adve & Hill ‘90]

- C++11 memory model [Boehm & Adve ‘08]
  - Data races $\rightarrow$ no semantics

- Java memory model [Manson et al. ‘05]
  - Data races $\rightarrow$ only type & memory safety preserved
  - But... the JMM is actually broken

“Nondeterminism is unavoidable, but data races are pure evil” [Boehm ‘12]
“Memory Models: A Case for Rethinking Parallel Languages and Hardware” [Adve & Boehm ‘10]
“The Case for System Support for Concurrency Exceptions” [Ceze et al. ‘09]
RS of synchronization-free regions [Ouyang et al. ’13]

Both racy and non-racy executions provide RS.

Region serializability of synchronization-free regions

Sequentially consistent
Atomicity

Operations appear to happen all at once or not at all

Serializability – execution equivalent to some serial execution of atomic blocks
Atomicity violation?

```c
int x = 1;

T1:
t = x;
t = t + 3;
x = t;

T2:
t = x;
t = t * 2;
x = t;
```
Atomicity violation?

```c
int x = 1;

T1:
    t = x;
    t = t + 3;
    x = t;

T2:
    t = x;
    t = t * 2;
    x = t;
```
Atomicity violation?

```java
int x = 1;

T1:
synchronized(m) { 
    t = x;
}

t = t + 3;

synchronized(m) { 
    x = t;
}

T2:

synchronized(m) { 
    t = x;
}

t = t * 2;

synchronized(m) { 
    x = t;
}
```
Atomicity

```java
int x = 1;

T1:

synchronized (m) {
    t = x;
    t = t + 3;
    x = t;
}

T2:

synchronized (m) {
    t = x;
    t = t * 2;
    x = t;
}
```
Atomicity

int x = 1;

T1:

synchronized (m) {
    t = x;
    t = t + 3;
    x = t;
}

T2:

synchronized (m) {
    t = x;
    t = t * 2;
    x = t;
}
Atomicity

```java
int x = 1;

T1:
synchronized (m) {
    t = x;
    t = t + 3;
    x = t;
}

T2:
synchronized (m) {
    t = x;
    t = t * 2;
    x = t;
}
```
Atomicity violation without data races

class Vector {
    synchronized boolean contains(Object o) {
        ...
    }
    synchronized void add(Object o) {
        ...
    }
}
Atomicity violation without data races

class Vector {
    synchronized boolean contains(Object o) { ... }  
    synchronized void add(Object o) { ... }
}

class Set {
    Vector vector;
    void add(Object o) {
        if (!vector.contains(o)) {
            vector.add(o);
        }
    }
}
Atomicity violation without data races

class Vector {
    synchronized boolean contains(Object o) { ... }
    synchronized void add(Object o) { ... }
}

class Set {
    Vector vector;
    synchronized void add(Object o) {
        if (!vector.contains(o)) {
            vector.add(o);
        }
    }
}
Atomicity violation without data races

class Vector {
    synchronized boolean contains(Object o) { ... }
    synchronized void add(Object o) { ... }
}

class Set {
    Vector vector;
    void add(Object o) {
        atomic {
            if (!vector.contains(o)) {
                vector.add(o);
            }
        }
    }
}
Concurrency exceptions?!

Java provides memory & type safety
- Buffer overflows, dangling pointers, array out-of-bounds, double frees, some memory leaks
- How are these handled? With exceptions?
Concurrency exceptions?!

Java provides memory & type safety

- Buffer overflows, dangling pointers, array out-of-bounds, double frees, some memory leaks
- How are these handled? With exceptions?

Should languages (and the runtime systems & hardware that support them) provide **concurrency** correctness?

Check & enforce: atomicity, strong memory model, determinism
Summary

General-purpose parallel software: hard & unsolved

Challenging semantics for parallel programs

Understand how to write correct, scalable programs
→ one of few experts
How Do Data Race Detectors Work?
Data Races

- Two accesses to same variable (one is a write)
- One access doesn’t happen before the other
  - Program order
  - Synchronization order
    - Acquire-release
    - Wait-notify
    - Fork-join
    - Volatile read-write
Data Races

- Two accesses to same variable (one is a write)
- One access doesn’t happen before the other
  - Program order
  - Synchronization order
    - Acquire-release
    - Wait-notify
    - Fork-join
    - Volatile read-write

Thread A
- write x

Thread B
- unlock m
Data Races

- Two accesses to same variable (one is a write)
- One access doesn’t happen before the other
  - Program order
  - Synchronization order
    - Acquire-release
    - Wait-notify
    - Fork-join
    - Volatile read-write

Thread A
- write x
- unlock m

Thread B
- lock m
- write x
Data Races

- Two accesses to same variable (one is a write)
- One access doesn’t happen before the other
  - Program order
  - Synchronization order
    - Acquire-release
    - Wait-notify
    - Fork-join
    - Volatile read-write

```
Thread A
write x
unlock m
read x

Thread B
lock m
write x
```
Data Races

- Two accesses to same variable (one is a write)
- One access doesn’t happen before the other
  - Program order
  - Synchronization order
    - Acquire-release
    - Wait-notify
    - Fork-join
    - Volatile read-write

**Thread A**
- write x
- unlock m
- read x

**Thread B**
- lock m
- write x

Race!
How do dynamic data race detectors work?

- Lockset: check a locking discipline
- Vector clocks: check happens-before
Tracks happens-before: sound & precise

- **80X** slowdown
- Each analysis step: $O(n)$ time  \( (n = \# \text{ of threads}) \)
Tracks happens-before: sound & precise
- 80X slowdown
- Each analysis step: $O(n)$ time  \hspace{1cm} (n = # of threads)

FastTrack  \hspace{1cm} [Flanagan & Freund ’09]
- Reads & writes (97%): $O(1)$ time
- Synchronization (3%): $O(n)$ time
- 8X slowdown
Vector Clock-Based Race Detection

- Tracks happens-before: sound & precise
  - 80X slowdown
  - Each analysis step: $O(n)$ time ($n = \# \text{ of threads}$)

- FastTrack  [Flanagan & Freund ’09]
  - Reads & writes (97%): $O(1)$ time
  - Synchronization (3%): $O(n)$ time
  - 8X slowdown

Problem today
Problem in future
Tracks happens-before: sound & precise
  - 80X slowdown
  - Each analysis step: $O(n)$ time
    (n = # of threads)

FastTrack [Flanagan & Freund ’09]
  - Reads & writes (97%): $O(1)$ time
  - Synchronization (3%): $O(n)$ time
  - 8X slowdown
Vector Clock-Based Race Detection

- **Thread A**
  - A: 5
  - B: 2

- **Thread B**
  - A: 3
  - B: 4

- Vector clocks
Vector Clock-Based Race Detection

**Thread A**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Thread B**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Thread A’s logical time

Thread B’s logical time

Vector clocks
Vector Clock-Based Race Detection

**Thread A**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Thread B**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Last logical time “received” from A

Last logical time “received” from B

Vector clocks
Vector Clock-Based Race Detection

Thread A

\[
\begin{array}{c|c}
A & B \\
5 & 2 \\
\end{array}
\]

Thread B

\[
\begin{array}{c|c}
A & B \\
3 & 4 \\
\end{array}
\]

Increment clock

unlock m

6 2

lock m
Vector Clock-Based Race Detection

Thread A

A B
5 2

Thread B

A B
3 4

unlock m

6 2

lock m

5 4

Join clocks
Vector Clock-Based Race Detection

Thread A

A  B
5  2

Thread B

A  B
3  4

unlock m

5  2

lock m

n = # of threads

O(n) time
Tracks happens-before: sound & precise
  - 80X slowdown
  - Each analysis step: \( O(n) \) time \((n = \# \text{ of threads})\)

- **FastTrack** [Flanagan & Freund ’09]
  - Reads & writes (97%): \( O(1) \) time
  - Synchronization (3%): \( O(n) \) time
  - 8X slowdown
Vector Clock-Based Race Detection

Thread A
- A: 5
- B: 2
- write x
- unlock m

Thread B
- A: 3
- B: 4
- lock m
- write x
- read x
Vector Clock-Based Race Detection

Thread A

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

write x

unlock m

Thread B

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

5@A

lock m

write x

read x
Vector Clock-Based Race Detection

Thread A

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

write x 5@A

unlock m

Thread B

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

lock m

write x

read x
Vector Clock-Based Race Detection

Thread A

\[
\begin{array}{c|c}
A & B \\
5 & 2 \\
\end{array}
\]

write x

unlock m

\[
\begin{array}{c|c}
A & B \\
6 & 2 \\
\end{array}
\]

Thread B

\[
\begin{array}{c|c}
A & B \\
3 & 4 \\
\end{array}
\]

write x

lock m

read x
Thread A

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

write x

unlock m

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Thread B

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

lock m

write x

read x
## Vector Clock-Based Race Detection

### Thread A
- **Write** $x$
- **Unlock** $m$
- **Read** $x$

### Thread B
- **Lock** $m$
- **Write** $x$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread A</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Thread B</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

5@A
Vector Clock-Based Race Detection

Thread A

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

write x

unlock m

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Happens before?

Thread B

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

lock m

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

write x

read x
Vector Clock-Based Race Detection

Thread A

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

write x

unlock m

| 6 | 2 |

Thread B

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

lock m

write x

read x
Vector Clock-Based Race Detection

Thread A

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

write x

unlock m

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Thread B

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Happens before?

lock m

write x

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

read x

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>@B</td>
<td></td>
</tr>
</tbody>
</table>
Vector Clock-Based Race Detection

Thread A

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

write x

unlock m

6 2

Thread B

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

lock m

write x

Happens before?

Race!

read x
<table>
<thead>
<tr>
<th></th>
<th><strong>FastTrack</strong> [Flanagan &amp; Freund ’09]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection rate</td>
<td>occurrence rate</td>
</tr>
<tr>
<td>Running time</td>
<td>$t(c_1 + c_2 n)$</td>
</tr>
</tbody>
</table>

No. of threads
### Accuracy & Performance

<table>
<thead>
<tr>
<th></th>
<th>FastTrack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection rate</td>
<td>occurrence rate</td>
</tr>
<tr>
<td>Running time</td>
<td>$t(c_1 + c_2n)$</td>
</tr>
</tbody>
</table>

- **Reads & writes**
- **Synchronization**
## Accuracy & Performance

<table>
<thead>
<tr>
<th></th>
<th>FastTrack [Flanagan &amp; Freund ’09]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection rate</td>
<td>occurrence rate</td>
</tr>
<tr>
<td>Running time</td>
<td>$t(c_1 + c_2n)$</td>
</tr>
</tbody>
</table>

- **Reads & writes**: Problem today
- **Synchronization**: Problem in future