Concurrency & Parallelism
CSE 3341

Reading: Chapter 12 in Scott
Outline

- **Motivation & examples**
- Threads, shared memory, & synchronization
  - How do locks work?
- Data races (a lower-level property)
- How do data race detectors work?
- Atomicity (a higher-level property)
- Concurrency exceptions & Summary

Extra:
- Double-checked locking
Hardware becoming more parallel
(\textit{more} instead of \textit{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
What are the possible behaviors?

```c
int x = 1;

T1:
    t = x;
    t = t + 1;
    x = t;

T2:
    t = x;
    t = t + 1;
    x = t;
```
Atomicity & determinism

```java
int x = 1;

T1:
synchronized (m) {
    t = x;
    t = t + 1;
    x = t;
}

T2:
synchronized (m) {
    t = x;
    t = t + 1;
    x = t;
}
```
What are the possible behaviors?

```
int x = 1;

T1:
t = x;
t = t + 1;
x = t;
```

```
T2:
t = x;
t = t * 2;
x = t;
```
Atomicity (still nondeterminism)

```java
int x = 1;

T1:
synchronized (m) {
    t = x;
    t = t + 1;
    x = t;
}

T2:
synchronized (m) {
    t = x;
    t = t * 2;
    x = t;
}
```
What are the possible behaviors?

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

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

**T2:**
```
if (flag) {
    int t = data;
    print(t);
}
```
Outline

- Motivation & examples
- Threads, shared memory, & synchronization
  - How do locks work?
- Data races (a lower-level property)
- How do data race detectors work?
- Atomicity (a higher-level property)
- Concurrency exceptions & Summary

Extra:
- Double-checked locking
Threads, shared memory, & locks

Each thread:
- has its own stack
- shares memory with other threads in same process (same virtual address space)

Compiler compiles code as though it were single-threaded!

Synchronization operations (e.g., lock acquire and release) order accesses to shared memory, providing:
- mutual exclusion
- ordering and visibility
Locks in Java

```java
int x = 1;

T1:

synchronized (m) {
    t = x;
    t = t + 1;
    x = t;
}

T2:

synchronized (m) {
    t = x;
    t = t * 2;
    x = t;
}
```
int x = 1;

T1:
acquire(m);
t = x;
t = t + 1;
x = t;
release(m);  

T2:
acquire(m);
t = x;
t = t * 2;
x = t;
release(m);
Locks in Java

```java
int x = 1;

T1:
acquire(m.lockBit);
    t = x;
    t = t + 1;
    x = t;
release(m.lockBit);
```

```java
T2:
acquire(m.lockBit);
    t = x;
    t = t * 2;
    x = t;
release(m.lockBit);
```

Locking (& other) bits
Locks in Java

```java
int x = 1;

T1:
while (m.lockBit != 0) {}
    m.lockBit = 1;
    t = x;
    t = t + 1;
    x = t;
    m.lockBit = 0;
```

```
T2:
while (m.lockBit != 0) {}
    m.lockBit = 1;
    t = x;
    t = t * 2;
    x = t;
    m.lockBit = 0;
```

Possible implementation of locks?
Locks in Java

```java
int x = 1;

T1:
while (!TAS(&m.lockBit,0,1)) {}  
t = x;
t = t + 1;
x = t;
m.lockBit = 0;

T2:
while (!TAS(&m.lockBit,0,1)) {}  
t = x;
t = t * 2;
x = t;
m.lockBit = 0;
```

Need an atomic operation like test-and-set (TAS)
Locks in Java

int x = 1;

T1:

while (!TAS(&m.lockBit,0,1)) {}  
    t = x;
    t = t + 1;
    x = t;
memory_fence;
m.lockBit = 0;

T2:

while (!TAS(&m.lockBit,0,1)) {}  
    t = x;
    t = t * 2;
    x = t;
memory_fence;
m.lockBit = 0;

• Fence needed for visibility (related to happens-before relationship, discussed later)
• Also: compiler obeys “roach motel” rules: can move operations into but not out of atomic blocks
Locks in Java

```java
int x = 1;

T1:

while (!TAS(&m.lockBit,0,1)) {
    t = x;
    t = t + 1;
    x = t;
    memory_fence;
    m.lockBit = 0;
}

T2:

while (!TAS(&m.lockBit,0,1)) {
    t = x;
    t = t * 2;
    x = t;
    memory_fence;
    m.lockBit = 0;
}
```

- Java locks are **reentrant**, so more than one bit is actually used (to keep track of nesting depth)
- Also, **spin (non-blocking) locks** are converted to **blocking locks** if there’s contention
Motivation & examples
Threads, shared memory, & synchronization
  ▪ How do locks work?
Data races (a lower-level property)
  ▪ How do data race detectors work?
  ▪ Atomicity (a higher-level property)
  ▪ Concurrency exceptions & Summary
Extra:
  ▪ Double-checked locking
Northeast Blackout of 2003
Northeast Blackout of 2003

50 million people
Northeast Blackout of 2003

- Energy Management System
  - Alarm and Event Processing Routine (1 MLOC)

Northeast Blackout of 2003

- Energy Management System
  - Alarm and Event Processing Routine (1 MLOC)

- Post-mortem analysis: 8 weeks
  “This fault was so deeply embedded, it took them weeks of poring through millions of lines of code and data to find it.” – Ralph DiNicola, FirstEnergy

Race condition
- Two threads writing to data structure simultaneously
- Usually occurs without error
  - Small window for causing data corruption

What is a data race?

- 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
Modern language memory models (via compiler+hardware) guarantee the following relationship:

Data race freedom $\Rightarrow$ Sequential consistency

However: Data race $\Rightarrow$ Weak or undefined semantics!

Sequential consistency: instructions appear to execute in an order that respect’s program order
Is there a data race?

Initially:

\[
\begin{align*}
\text{int } & \text{ data } = 0; \\
\text{boolean } & \text{ flag } = \text{false};
\end{align*}
\]

**T1:**

\[
\begin{align*}
\text{data } & = 42; \\
\text{flag } & = \text{true};
\end{align*}
\]

**T2:**

\[
\begin{align*}
\text{if (flag)} & \\
\text{t } & = \text{data;}
\end{align*}
\]
Is there a data race?

Initially:

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

T1:

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

T2:

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

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

T1:
flag = true;

T2:
if (flag)
t = data;

data = 42;
Possible behavior

Initially:

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

**T1:**

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

**T2:**

```java
t2 = data;
```

```java
if (flag)
    t = t2;
```
Is there a data race?

Initially:

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

**T1:**

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

**T2:**

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

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

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

Happens-before relationship

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

Initially:

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

**T1:**

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

**T2:**

```
if (flag)
    ... = data;
```
What can this program print?

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

T1:
data = 42;
flag = true;

T2:
while (!flag) { }
print(data);
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:
int data = 0;
boolean flag = false;

T1:
flag = true;
data = 42;

T2:
boolean f;
do {
  f = flag;
} while (!f);
int d = data;
print(d);
What can this program print?

Initially:

\[
\begin{align*}
\text{int data} &= 0; \\
\text{boolean flag} &= \text{false};
\end{align*}
\]

\textbf{T1:}

\[
\begin{align*}
data &= 42; \\
\text{flag} &= \text{true};
\end{align*}
\]

\textbf{T2:}

\[
\begin{align*}
\text{int d} &= \text{data}; \\
\text{boolean f} &= \text{false}; \\
\text{do} & \{
\text{f} = \text{flag};
\} \text{ while } (!f); \\
\text{print(d)};
\end{align*}
\]
Outline

- Motivation & examples
- Threads, shared memory, & synchronization
  - How do locks work?
- Data races (a lower-level property)
- How do data race detectors work?
- Atomicity (a higher-level property)
- Concurrency exceptions & Summary

Extra:
- Double-checked locking
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

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>write x</td>
<td>lock m</td>
</tr>
<tr>
<td>unlock m</td>
<td>write x</td>
</tr>
</tbody>
</table>
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

Thread B

- lock m
- write x

Race!
Possible Strategies

How do dynamic data race detectors work?

- Vector clocks: check happens-before
- Lockset: assume and check locking discipline
- Other strategies
What is a Vector Clock?

- Each thread T maintains its own logical clock ‘c’
  - Initially, c = 0 when T starts
  - Incremented at synchronization release operations
    - unlock m, volatile write, Thread.join()

- Vector clock is a vector of logical clocks
Vector Clock and Happens Before

\[ V_1 < V_2 \text{ iff } \forall t \ V_1(t) \leq V_2(t) \]

\[
\begin{array}{ccc}
A & B & C \\
1 & 2 & 3 \\
\end{array}
\begin{array}{ccc}
A & B & C \\
5 & 6 & 7 \\
\end{array} \quad \begin{array}{ccc}
true \\
\end{array}

\[
\begin{array}{ccc}
A & B & C \\
4 & 5 & 3 \\
\end{array}
\begin{array}{ccc}
A & B & C \\
4 & 5 & 3 \\
\end{array} \quad \begin{array}{ccc}
true \\
\end{array}

\[
\begin{array}{ccc}
A & B & C \\
4 & 5 & 3 \\
\end{array}
\begin{array}{ccc}
A & B & C \\
4 & 2 & 3 \\
\end{array} \quad \begin{array}{ccc}
false \\
\end{array}

Vector Clock-Based Race Detection

Thread A

A | B
5 | 2

Thread B

A | B
3 | 4

Vector clocks
Vector Clock-Based Race Detection

Thread A

Thread B

Vector clocks

Thread A’s logical time

Thread B’s logical time
Vector Clock-Based Race Detection

Thread A

5  2

Thread B

3  4

Last logical time “received” from A

Last logical time “received” from B

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>

write x

Thread B

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Vector Clock-Based Race Detection

Thread A

A  B
5  2

write x

unlock m

5  2

6  2

Thread B

A  B
3  4

Increment clock
Vector Clock-Based Race Detection

Thread A:
- Write x
- Unlock m

Thread B:
- Lock m

Increment clock
Vector Clock-Based Race Detection

Thread A

Thread B

Increment clock

Join clocks

write x
unlock m
lock m
Vector Clock-Based Race Detection

Thread A
- A: 5
- B: 2
- Write x
- Unlock m

Thread B
- A: 3
- B: 4
- Lock m

\[ n = \# \text{ of threads} \]

\[ \text{O}(n) \text{ time} \]
Vector Clock-Based Race Detection

Thread A

- 5 2
- write x
- 6 2
- unlock m

Thread B

- 3 4
- lock m
- 5 4
- write x
Vector Clock-Based Race Detection

Thread A

5 2
write x

6 2
unlock m

Thread B

3 4
lock m

5 4
write x

Happens before?
Vector Clock-Based Race Detection

Thread A

A B
5 2

write x

unlock m

A B
6 2

Thread B

A B
3 4

lock m

A B
5 4

Race!

read x

write x

A B
5 4
Soundness and Precision

- **Soundness**
  - No false negatives, i.e., all data races in *that* execution will be reported.

- **Precision**
  - No false positives, i.e., all reports are true data races.

---

These are not standard terms across all domains, e.g., architects might refer to these properties as complete and sound.
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
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
FastTrack’s Insight

- In a data-race-free (DRF) program, writes are totally ordered
  - Can maintain only the “last” writer
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>

lock m

write x

read x
## Vector Clock-Based Race Detection

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

**Thread A:**
- **Write x:** 5@A
- **Unlock m**

**Thread B:**
- **Lock m**
- **Write x**
- **Read x**
Vector Clock-Based Race Detection

Thread A

A  B
5  2

write  x

unlock  m

Thread B

A  B
3  4

lock  m

write  x

read  x
## Vector Clock-Based Race Detection

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

- **Thread A**
  - **Write x**
    - 5@A
  - **Unlock m**
    - 6 2

- **Thread B**
  - **Lock m**
  - **Write x**
  - **Read x**
Vector Clock-Based Race Detection

Thread A

Thread B

A   B
5   2

write x

A   B
3   4

unlock m

lock m

write x

read x
Vector Clock-Based Race Detection

Thread A
A  B
5  2
write x
unlock m
6  2

Thread B
A  B
3  4
write x
lock m
5  4
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>

write x

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

<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

Happens before?

5@A
Vector Clock-Based Race Detection

Thread A

A | B
---|---
5  | 2

write x

unlock m

A | B
---|---
6  | 2

Thread B

A | B
---|---
3  | 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>

Thread B

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

write x

unlock m

read x

write x

lock m

Happens before?
Vector Clock-Based Race Detection

Thread A

Thread B

write x

unlock m

lock m

write x

Happens before?

Race!
What is a Lockset?

- Keeps track of the locks associated with each thread and program variable
What is a Lockset?

- Keeps track of the locks associated with each thread and program variable

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Lockset_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock m</td>
<td>L = {m}</td>
</tr>
<tr>
<td>write x</td>
<td>L = {m}</td>
</tr>
<tr>
<td>lock n</td>
<td>L = {m, n}</td>
</tr>
<tr>
<td>write y</td>
<td>L = {m, n}</td>
</tr>
<tr>
<td>unlock n</td>
<td>L = {m}</td>
</tr>
<tr>
<td>unlock m</td>
<td>L = {m}</td>
</tr>
<tr>
<td>read x</td>
<td>L = {m}</td>
</tr>
</tbody>
</table>
Lockset Algorithms

- Two accesses from different threads with non-intersecting locksets form a data race
- Can detect more data races than happens-before tracking
  - A property referred to as coverage
Is there a data race on variable y?

Thread A

\[ y = y + 1 \]

lock m

\[ v = v + 1 \]

unlock m

Happens before

Thread B

lock m

\[ v = v + 1 \]

unlock m

\[ y = y + 1 \]
Is there a data race on variable y?

Thread A

\[ y = y + 1 \]
\[ \text{lock } m \]
\[ v = v + 1 \]
\[ \text{unlock } m \]

Thread B

\[ \text{lock } m \]
\[ v = v + 1 \]
\[ \text{unlock } m \]
\[ y = y + 1 \]

No data race with happens-before
Is there a data race on variable y?
Alternate interleaving

Thread A

\[ y = y + 1 \]
lock m
\[ v = v + 1 \]
unlock m

Thread B

lock m
\[ v = v + 1 \]
unlock m
\[ y = y + 1 \]
Is there a data race on variable y?
Alternate interleaving

Thread A

y = y + 1
lock m
v = v + 1
unlock m

Happens before?

Thread B

lock m
v = v + 1
unlock m
y = y + 1

Data race with happens-before

Data race with lockset
## Observed Interleaving

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y = y + 1 )</td>
<td>( v = v + 1 )</td>
</tr>
<tr>
<td>( \text{lock } m )</td>
<td>( \text{lock } m )</td>
</tr>
<tr>
<td>( v = v + 1 )</td>
<td>( v = v + 1 )</td>
</tr>
<tr>
<td>( \text{unlock } m )</td>
<td>( \text{unlock } m )</td>
</tr>
</tbody>
</table>

No data race with happens-before

Happens before
Thread A

\[ y = y + 1 \]

lock m

\[ v = v + 1 \]

unlock m

Happens before

Thread B

\[ v = v + 1 \]

unlock m

\[ y = y + 1 \]

Data race with lockset
## Data Race on y with Lockset

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Lockset</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = y + 1</td>
<td>lock m</td>
<td>$L_y = {}$</td>
</tr>
<tr>
<td>lock m</td>
<td>v = v + 1</td>
<td>$L_v = {m}$</td>
</tr>
<tr>
<td>unlock m</td>
<td>lock m</td>
<td>$L_v = {m}$</td>
</tr>
<tr>
<td>v = v + 1</td>
<td>unlock m</td>
<td>$L_v = {m}$</td>
</tr>
<tr>
<td>y = y + 1</td>
<td></td>
<td>$L_y = {}$</td>
</tr>
<tr>
<td>Thread A</td>
<td>Thread B</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td>$x = \text{new Object}();$</td>
<td>$\text{while (!flag);} }$</td>
<td></td>
</tr>
<tr>
<td>$\text{flag} = \text{true};$</td>
<td>$x.\text{print}();$</td>
<td></td>
</tr>
</tbody>
</table>

volatile boolean flag = false;
volatile boolean flag = false;

Thread A

x = new Object();
flag = true;

Thread B

while (!flag);

while (!flag);
x.print();

B is blocked

Happens before
Happens Before vs Lockset

- Lockset algorithms
  - Generally unsound and imprecise
  - Better coverage
- Vector clock algorithms
  - Sound and precise
  - Limited coverage
Outline

- Motivation & examples
- Threads, shared memory, & synchronization
  - How do locks work?
- Data races (a lower-level property)
- How do data race detectors work?
- Atomicity (a higher-level property)
- Concurrency exceptions & Summary

Extra:
- Double-checked locking
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 + 1;
    x = t;

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

```c
int x = 1;

T1:
  t = x;
  t = t + 1;
  x = t;

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

```java
int x = 1;

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

   t = t + 1;

synchronized(m) {
    x = t;
}

T2:
synchronized(m) {
    t = x;
}

   t = t * 2;

synchronized(m) {
    x = t;
}

Still an atomicity violation
```
Atomicity

```
int x = 1;

T1:
synchronized (m) {
    t = x;
    t = t + 1;
    x = t;
}

T2:
synchronized (m) {
    t = x;
    t = t * 2;
    x = t;
}
```
int x = 1;

T1:

synchronized (m) {
    t = x;
    t = t + 1;
    x = t;
}

T2:

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

```java
int x = 1;

T1: synchronized (m) {
    t = x;
    t = t + 1;
    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) { ... }
}
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);
            }
        }
    }
}
Outline

- Motivation & examples
- Threads, shared memory, & synchronization
  - How do locks work?
- Data races (a lower-level property)
- How do data race detectors work?
- Atomicity (a higher-level property)
- Concurrency exceptions & Summary

Extra:
- Double-checked locking
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?
Concurrent 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, SC/DRF, determinism
Summary

General-purpose parallel software: hard & unsolved

Challenging semantics for parallel programs

Understand how to write correct, scalable programs
→ one of few experts
Outline

- Motivation & examples
- Threads, shared memory, & synchronization
  - How do locks work?
- Data races (a lower-level property)
- How do data race detectors work?
- Atomicity (a higher-level property)
- Concurrency exceptions & Summary

Extra:
- Double-checked locking
class Movie {
    Vector<String> comments;

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

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

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

```java
class MyClass {
    private Vector comments;

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

```java
class MyClass {
    private Vector comments;

    public void addComment(String s) {
        if (comments == null) {
            comments = null;
        }
        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);
        }
    } else {
        comments.add(s);
    }
}
```