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

"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

Hardware becoming more parallel
(more instead of faster cores)
Software must become more parallel

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?

```
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;
    }
```

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?

```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?

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

T1:
    data = 42;
    flag = true;

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

Initially:

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

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

int x = 1;

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

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

T1: acquire(m);
T2: acquire(m);

while (m.lockBit != 0) {} 
T1: m.lockBit = 1;
T2: m.lockBit = 1;

t = x;
T1: t = t + 1;
T2: t = t * 2;

x = t;
T1: x = t;
T2: x = t;

m.lockBit = 0;
T1: m.lockBit = 0;
T2: m.lockBit = 0;

memory_fence;
T1: memory_fence;
T2: memory_fence;

Locking (& other) bits

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

• 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

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

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

DRF0-based memory models
Modern language memory models (via compiler+hardware) guarantee the following relationship:
Data race freedom → Sequential consistency
However: Data race → Weak or undefined semantics!
Sequential consistency: instructions appear to execute in an order that respects program order

Is there a data race?
Initially:
- int data = 0;
- boolean flag = false;

T1:  
data = 42;
flag = true;
if (flag)  
t = data;

T2:  
data = 42;
flag = true;
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:
int data = 0;
boolean flag = false;

T1:

data = 42;
flag = true;

T2:
t2 = data;

if (flag)
t = t2;

Is there a data race?

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

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

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 = ...;

T2:

boolean f;
synchronized (m) {
    flag = true;
}
synchronized (m) {
    f = flag;
    if (f)
        ... = data;

Is there a data race?

Initially:
int data = 0;

T1:
data = 42;
flag = true;

T2:

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

T1:
data = 42;
flag = true;

T2:

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

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

Thread T1:
- `flag = true;`
- `data = 42;`

Thread T2:
- `boolean f;`
  - `do { f = flag; } while (!f);`
- `int d = data;`
- `print(d);`

What can this program print?

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

Thread T1:
- `data = 42;`
- `flag = true;`

Thread T2:
- `int d = data;`
- `boolean f;`
  - `do { f = flag; } while (!f);`
- `print(d);`

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

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

Possible Strategies

How do dynamic data race detectors work?
- Vector clocks: check happens-before
- Lockset: assume and check locking discipline
- Other strategies

Vector Clock and Happens Before

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>( V_1 )</th>
<th>( V_2 )</th>
<th>Happens Before</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>( 1 )</td>
<td>( 2 )</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
<td>( 4 )</td>
<td>( 5 )</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
<td>( 4 )</td>
<td>( 2 )</td>
<td>false</td>
</tr>
</tbody>
</table>

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-Based Race Detection

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>( V_1 )</th>
<th>( V_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>Vector clocks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Clock-Based Race Detection

Thread A

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

Thread B

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

Last logical time “received” from A

Last logical time “received” from B

Thread A’s logical time

Thread B’s logical time

Vector clocks

Thread A

\[
\begin{array}{c}
5 \ 2 \\
\end{array}
\]

Thread B

\[
\begin{array}{c}
3 \ 4 \\
\end{array}
\]

Write x

Unlock m

Increment clock

Join clocks

Thread A

\[
\begin{array}{c}
5 \ 2 \\
\end{array}
\]

Thread B

\[
\begin{array}{c}
3 \ 4 \\
\end{array}
\]

Write x

Unlock m

Increment clock

Lock m

Thread A

\[
\begin{array}{c}
5 \ 2 \\
\end{array}
\]

Thread B

\[
\begin{array}{c}
3 \ 4 \\
\end{array}
\]

Write x

Unlock m

Increment clock

Lock m

Join clocks
Vector Clock-Based Race Detection

Thread A

<table>
<thead>
<tr>
<th>5 2</th>
<th>write x</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 2</td>
<td>unlock m</td>
</tr>
</tbody>
</table>

Thread B

<table>
<thead>
<tr>
<th>3 4</th>
<th>lock m</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 4</td>
<td>O(n) time</td>
</tr>
</tbody>
</table>

\( n = \# \text{of threads} \)

Vector Clock-Based Race Detection

Thread A

<table>
<thead>
<tr>
<th>5 2</th>
<th>write x</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 2</td>
<td>unlock m</td>
</tr>
</tbody>
</table>

Thread B

<table>
<thead>
<tr>
<th>3 4</th>
<th>lock m</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 4</td>
<td>write x</td>
</tr>
</tbody>
</table>

Happens before?

Vector Clock-Based Race Detection

Thread A

<table>
<thead>
<tr>
<th>5 2</th>
<th>write x</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 2</td>
<td>unlock m</td>
</tr>
</tbody>
</table>

Thread B

<table>
<thead>
<tr>
<th>3 4</th>
<th>lock m</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 4</td>
<td>write x</td>
</tr>
</tbody>
</table>

Happens before?

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.

Soundness and Precision

- 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
  - \(80\times\) 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
  - \(8\times\) 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**

\[
\begin{array}{c}
A & B \\
5 & 2 \\
\text{write } x \\
\text{unlock } m \\
\text{lock } m \\
\text{write } x \\
\text{read } x
\end{array}
\]

**Thread B**

\[
\begin{array}{c}
A & B \\
3 & 4 \\
\text{write } x \\
\text{unlock } m \\
\text{lock } m \\
\text{write } x \\
\text{read } x
\end{array}
\]
Vector Clock-Based Race Detection

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

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

Happens before?

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

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

Happens before?

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

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

Happens before?

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

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

Happens before?
What is a Lockset?

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

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

\[
\begin{align*}
\text{y} &= \text{y} + 1 \\
\text{lock m} &\quad \text{(Happens before)} \\
\text{v} &= \text{v} + 1 \\
\text{unlock m} &
\end{align*}
\]

Thread B

\[
\begin{align*}
\text{lock m} &
\end{align*}
\]

No data race with happens-before

Is there a data race on variable y? Alternate interleaving

Thread A

\[
\begin{align*}
\text{lock m} &
\end{align*}
\]

Thread B

\[
\begin{align*}
\text{lock m} &
\end{align*}
\]

\[
\begin{align*}
\text{y} &= \text{y} + 1 \\
\text{unlock m} &
\end{align*}
\]
**Is there a data race on variable y? Alternate interleaving**

- **Thread A**
  - y = y + 1
  - lock m
  - v = v + 1
  - unlock m

- **Thread B**
  - y = y + 1
  - lock m
  - v = v + 1
  - unlock m

**Observed Interleaving**

- **Thread A**
  - y = y + 1
  - lock m
  - v = v + 1
  - unlock m

- **Thread B**
  - y = y + 1
  - lock m
  - v = v + 1
  - unlock m

**Data Race on y with Lockset**

- **Thread A**
  - y = y + 1
  - lock m
  - v = v + 1
  - unlock m
  - y = y + 1

- **Thread B**
  - y = y + 1
  - lock m
  - v = v + 1
  - unlock m
  - y = y + 1

**Lockset Algorithms are Imprecise**

```
volatile boolean flag = false;

Thread A
x = new Object();
flag = true;
while (!flag);
x.print();
```

```
Thread B
x = new Object();
flag = true;
while (!flag);
x.print();
```

B is blocked
**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?**

```
int x = 1;

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

T2:  
    t = x;
    t = t * 2;
    x = t;
```

Still an atomicity violation

```
int x = 1;

T1:  
synchronized(m) {
    t = x;
    t = t + 1;
    x = t;
}  
synchronized(m) {
    t = x;
    t = t * 2;
    x = t;
}

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

```
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

```
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 Set {
   Vector vector;
   void add(Object o) {
      if (!vector.contains(o)) {
         vector.add(o);
      }
   }
}
```
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?

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, 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
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);
            }
        }
        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);
            }
        }
        comments.add(s);
    }
}

Another example: double-checked locking

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

Another example: double-checked locking

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