Concurrency and Parallelism
CSE 3341

Reading:
Chapter 13 in 4th Edition

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

Motivation

- A program is **concurrent** if it may have more than one active execution context – “threads”
  - A **thread** is the smallest sequence of programmed instructions that can be managed independently by the OS scheduler
- We do this to
  - Capture the logical structure of a program
  - Exploit parallelism for speed
  - Cope with physical distribution

Concurrent vs Parallel vs Distributed

- A concurrent system is **parallel** if more than one task can be physically active at once (this requires >1 processor)
- A parallel system is **distributed** if processors/devices are physically separated in the real world
- Parallelism vs switching is a performance and implementation issue, not a semantic one

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

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

```
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)

```
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

```
int x = 1;
```

T1:

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

T2:

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

Locks in Java

```
int x = 1;
```

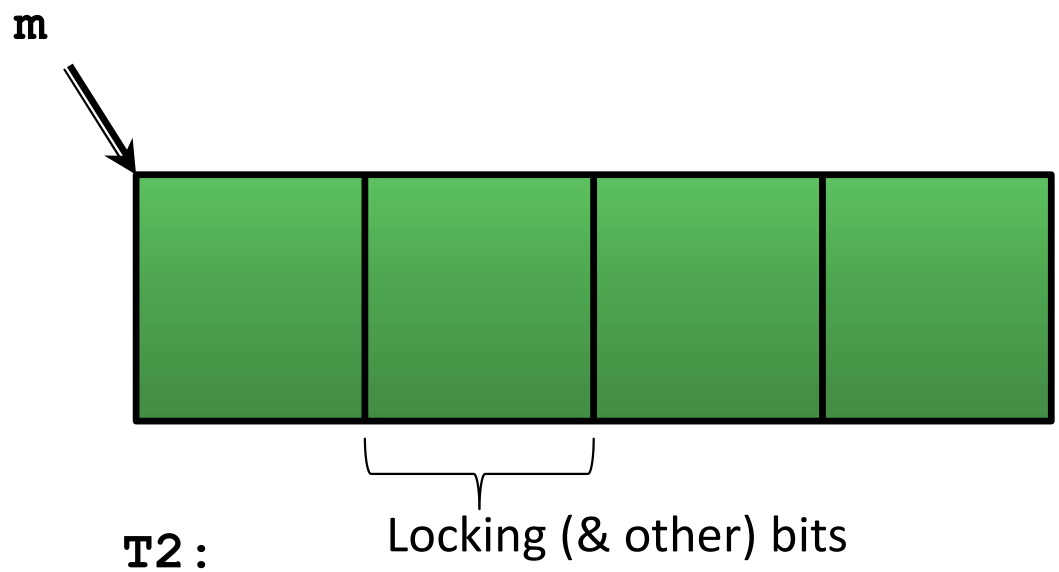
T1:

```
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




```
int x = 1;
```

T1:

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

T2:

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

Locks in 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

```
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

```
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

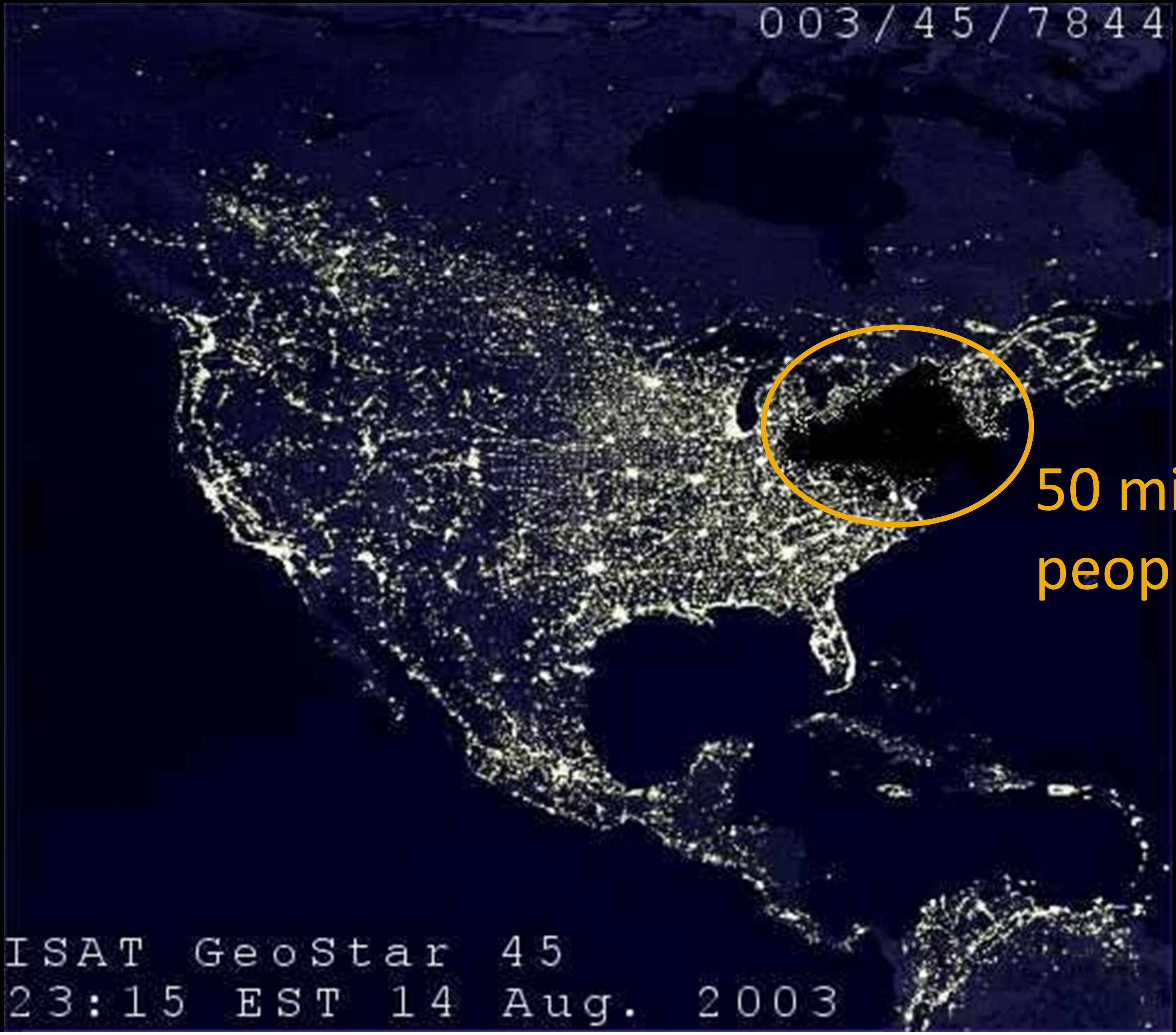
- Motivation & examples
- Threads, shared memory, & synchronization
  - How do locks work?
- **Data races (a lower-level property)**
- How do data race detectors work?
- Atomicity (a higher-level property)
- Concurrency exceptions & Summary

Extra:

- Double-checked locking

Northeast Blackout of 2003

Northeast Blackout of 2003



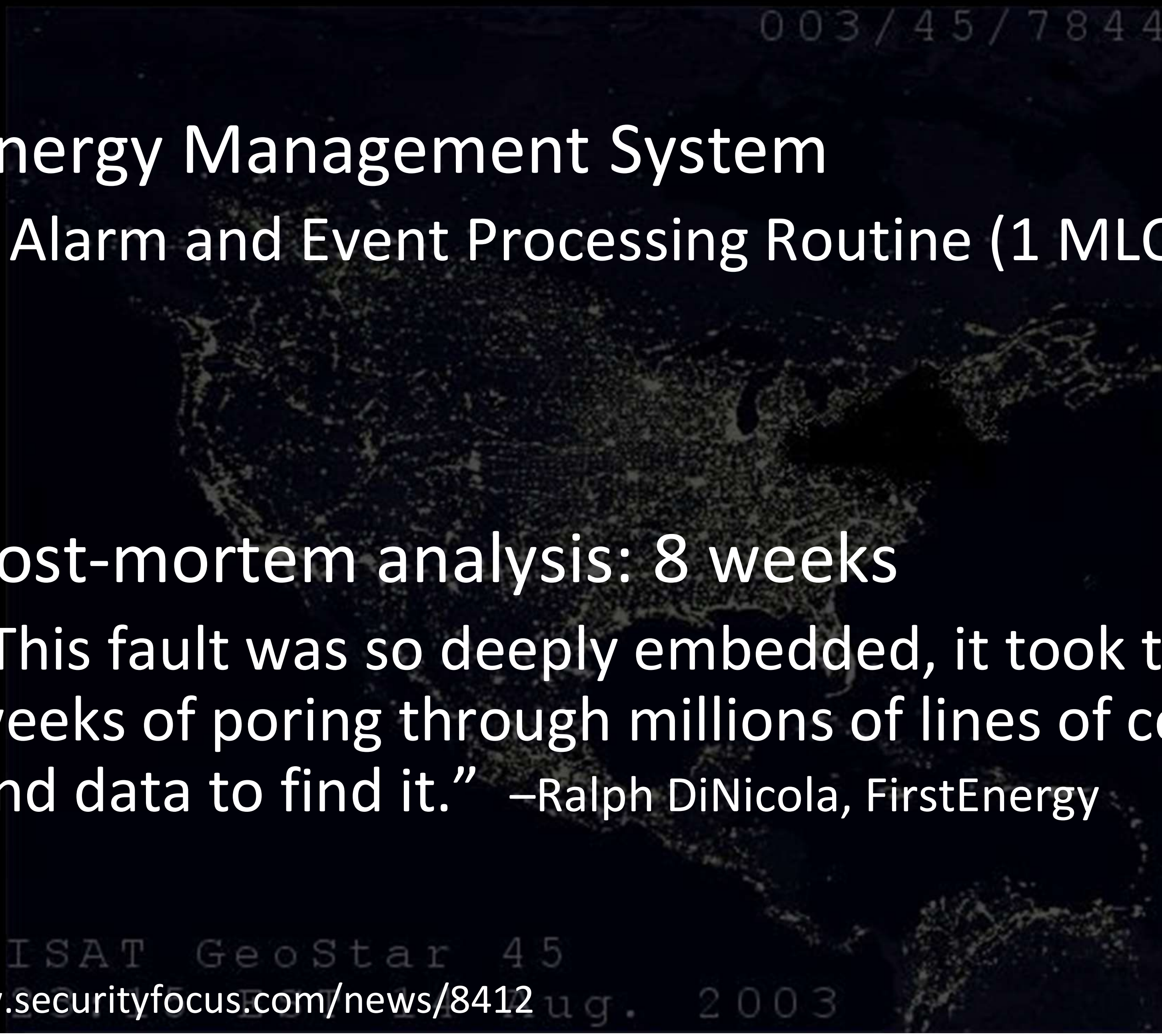
50 million people

Northeast Blackout of 2003

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

http://www.securityfocus.com/news/8412

Northeast Blackout of 2003



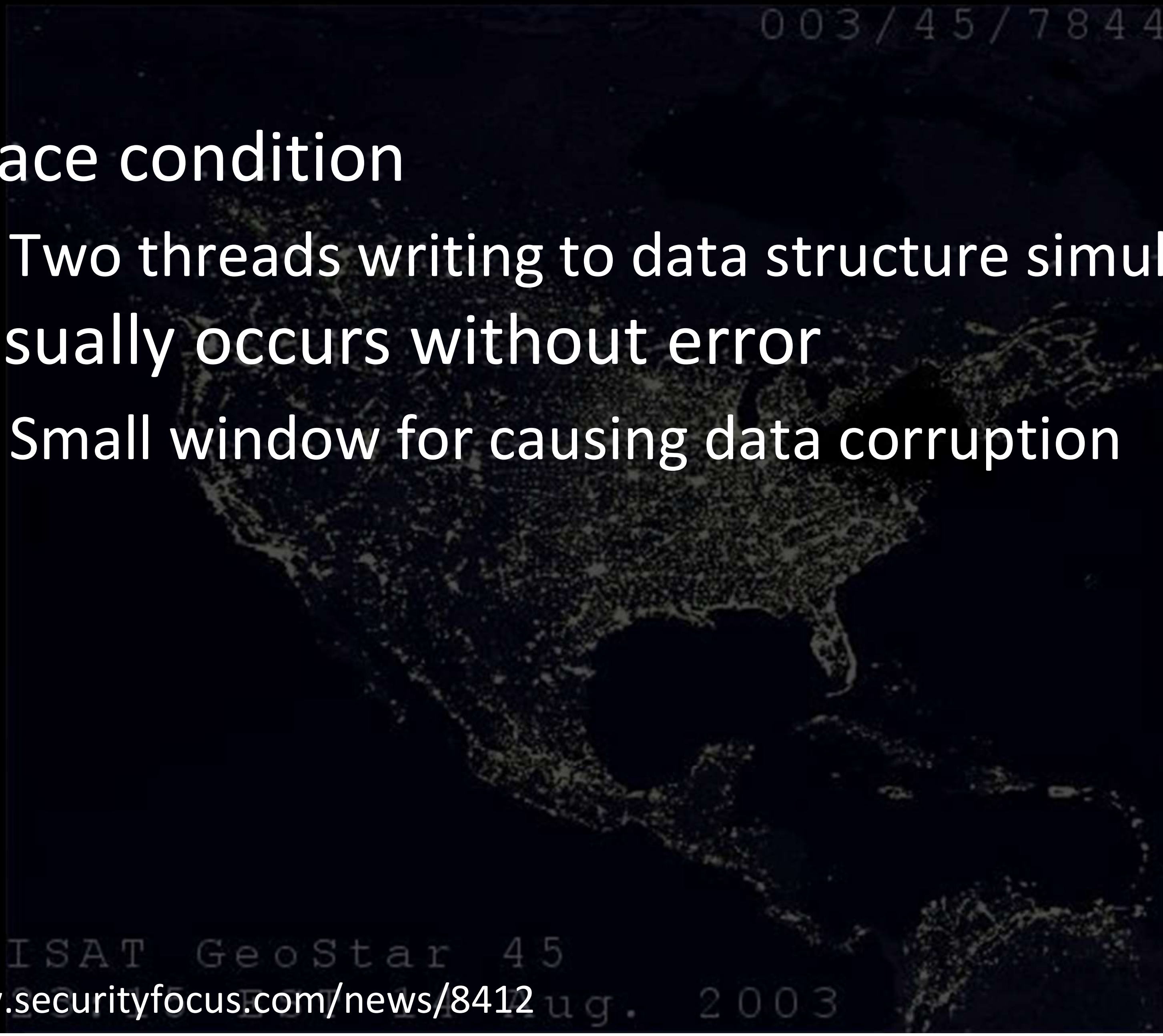
- Energy Management System
  - Alarm and Event Processing Routine (1 MLOC)

- 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

http://www.securityfocus.com/news/8412

Northeast Blackout of 2003



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

http://www.securityfocus.com/news/8412

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;
```

T2:

```
if (flag)
  t = data;
```

Is there a data race?

Initially:

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

T1:

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

T2:

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

T2:

```
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;
```

Happens-before relationship

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:

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

T1:

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

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;
```

T1:

```
flag = true;

data = 42;
```

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;
```

T1:

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

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

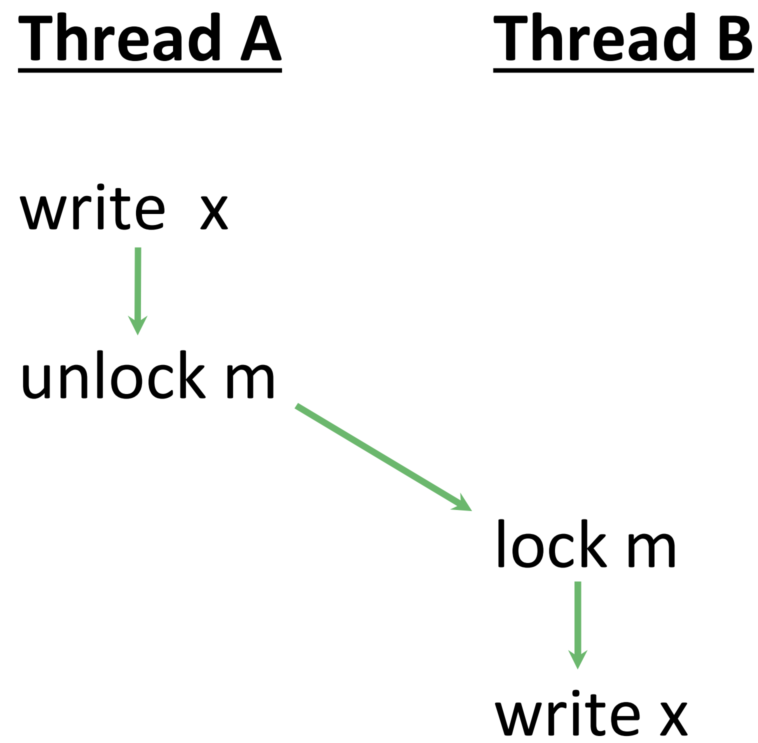
Data Races

- 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	**Thread B**
write x	
↓	
unlock m	

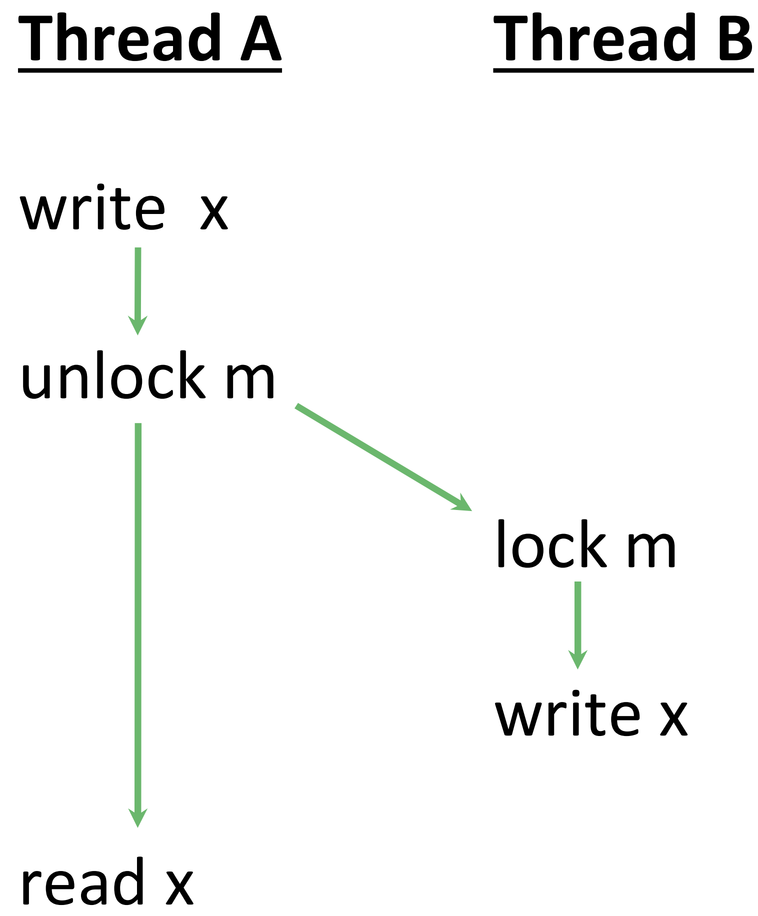
Data Races

- 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



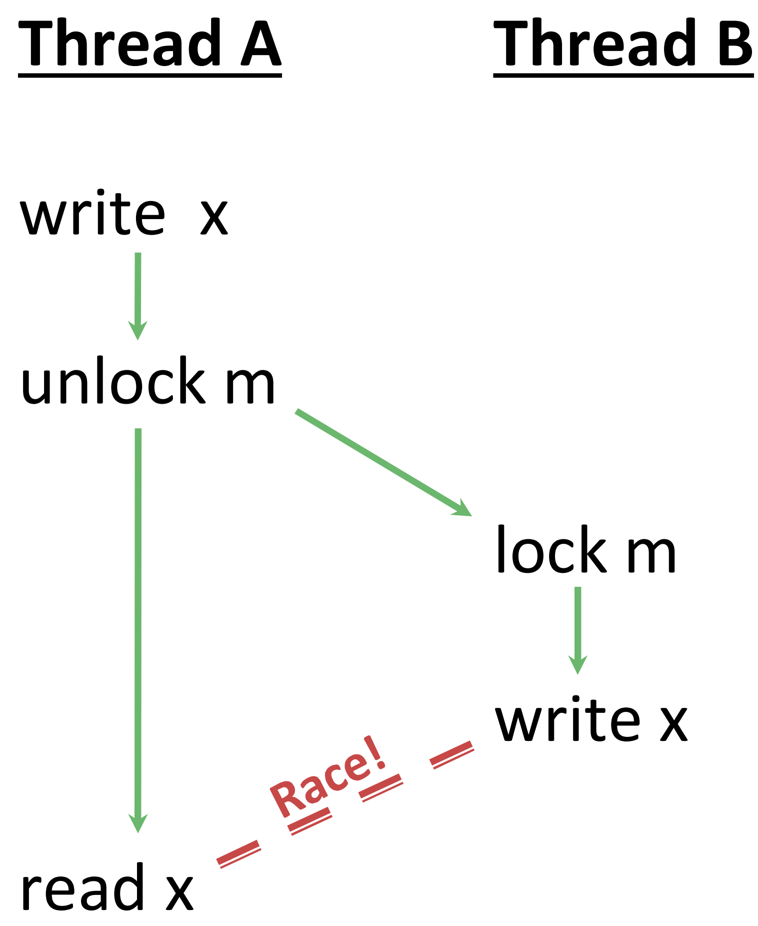
Data Races

- 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



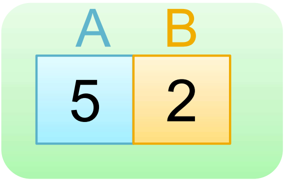
Data Race Detection: Possible Strategies

How do dynamic data race detectors work?

- Vector clocks: check happens-before
- Collision analysis: simultaneous conflicting accesses
- Lockset: assume and check locking discipline

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
- Vector clock is a vector of logical clocks



Vector Clock and Happens Before

$$V_1 \sqsubseteq V_2 \quad \text{iff} \quad \forall t \;\; V_1(t) \le V_2(t)$$

$\begin{array}{|c|c|c|} \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} \\ \hline 1 & 2 & 3 \\ \hline \end{array} \sqsubseteq \begin{array}{|c|c|c|} \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} \\ \hline 5 & 6 & 7 \\ \hline \end{array}$ **true**

$\begin{array}{|c|c|c|} \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} \\ \hline 4 & 5 & 3 \\ \hline \end{array} \sqsubseteq \begin{array}{|c|c|c|} \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} \\ \hline 4 & 5 & 3 \\ \hline \end{array}$ **true**

$\begin{array}{|c|c|c|} \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} \\ \hline 4 & 5 & 3 \\ \hline \end{array} \sqsubseteq \begin{array}{|c|c|c|} \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} \\ \hline 4 & 2 & 3 \\ \hline \end{array}$ **false**

Vector Clock-Based Race Detection

Note:

Starting the Vector Clock algorithm, I would initialize the clocks to [1, 0] and [0, 1].

The intuition here is that Thread A knows it has started execution, but does not know that thread B has started execution.

When a thread acquires the lock, information is gained from the last thread that held the lock. On the first lock, no information is gained.

The next few slides are meant to represent the state after several locks/unlocks, so the vector clocks have advanced significantly.

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

A	B
5	2

Thread B

A	B
3	4

Vector clocks

Thread A's logical time

Thread B's logical time

Vector Clock-Based Race Detection

Thread A

A	B
5	2

Thread B

A	B
3	4

Vector clocks

Last logical time “received” from A

Last logical time “received” from B

Vector Clock-Based Race Detection

Thread A

A	B
5	2

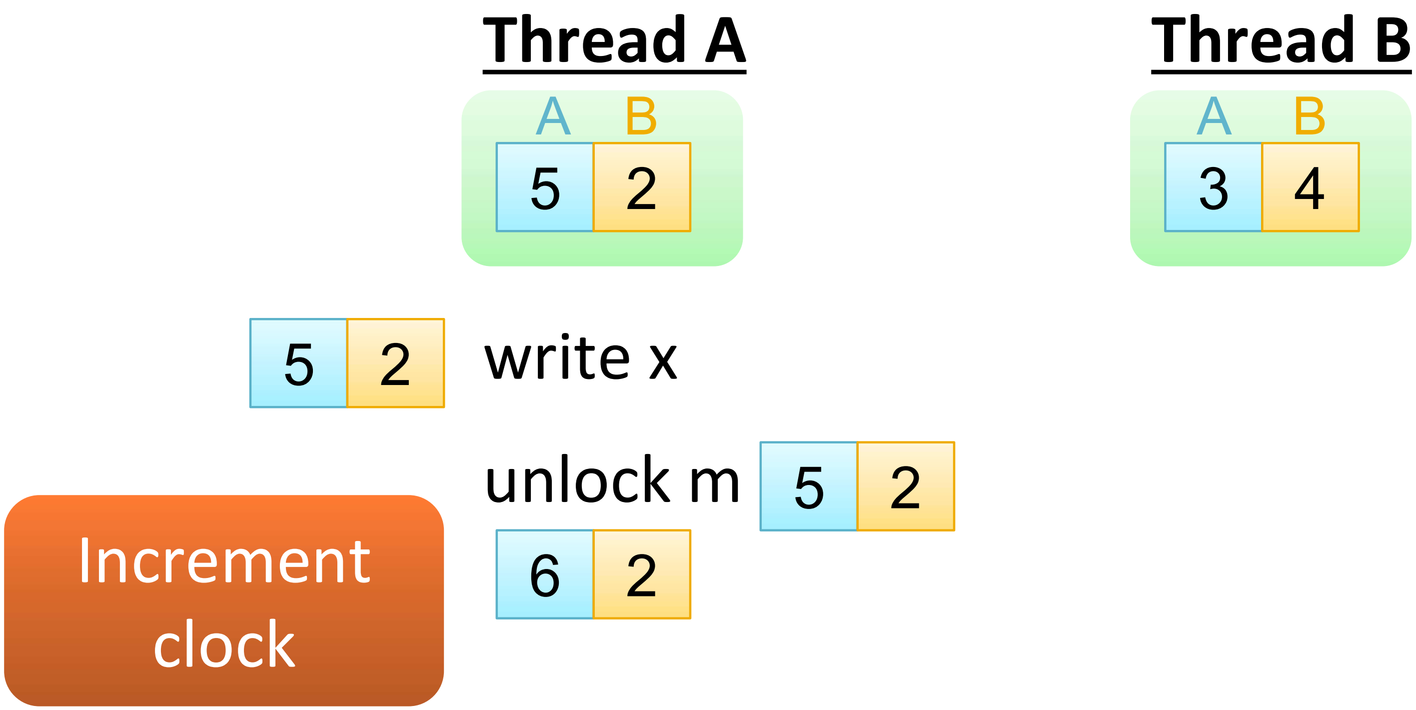
Thread B

A	B
3	4

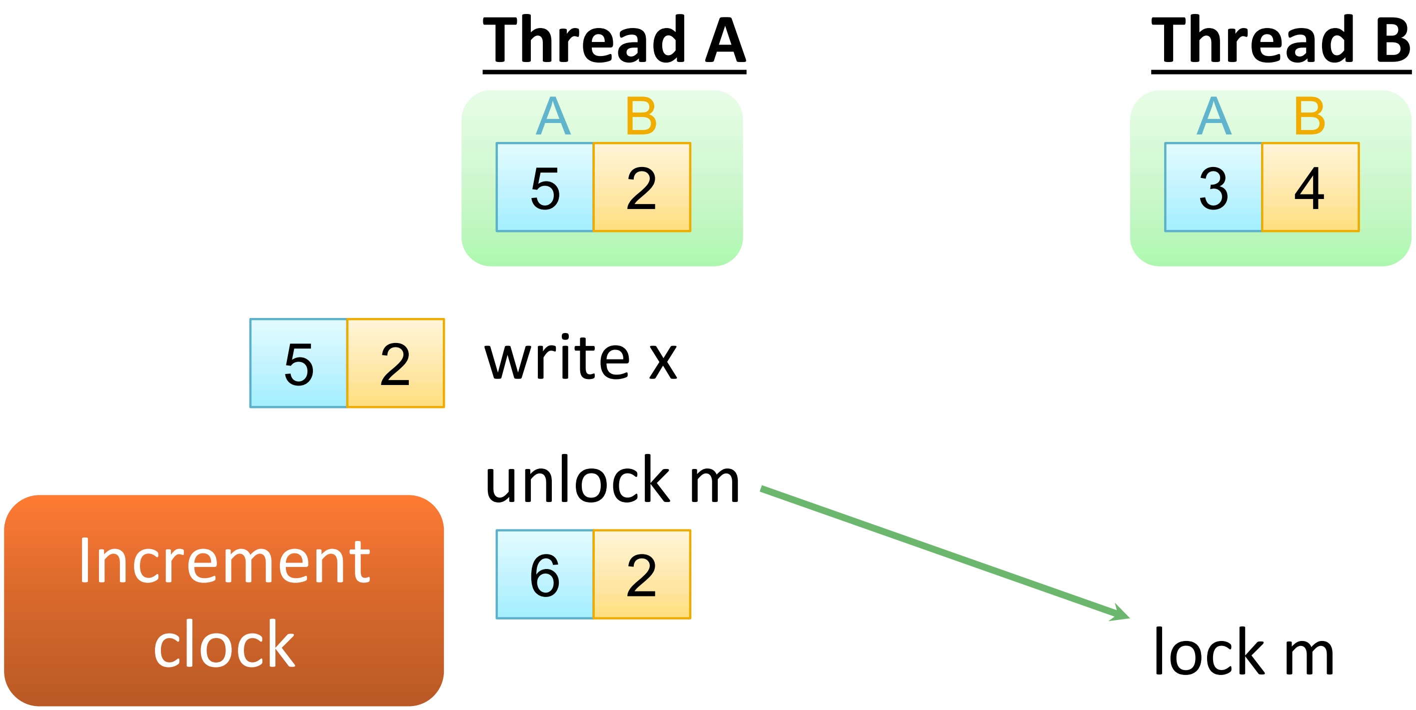
5	2

write x

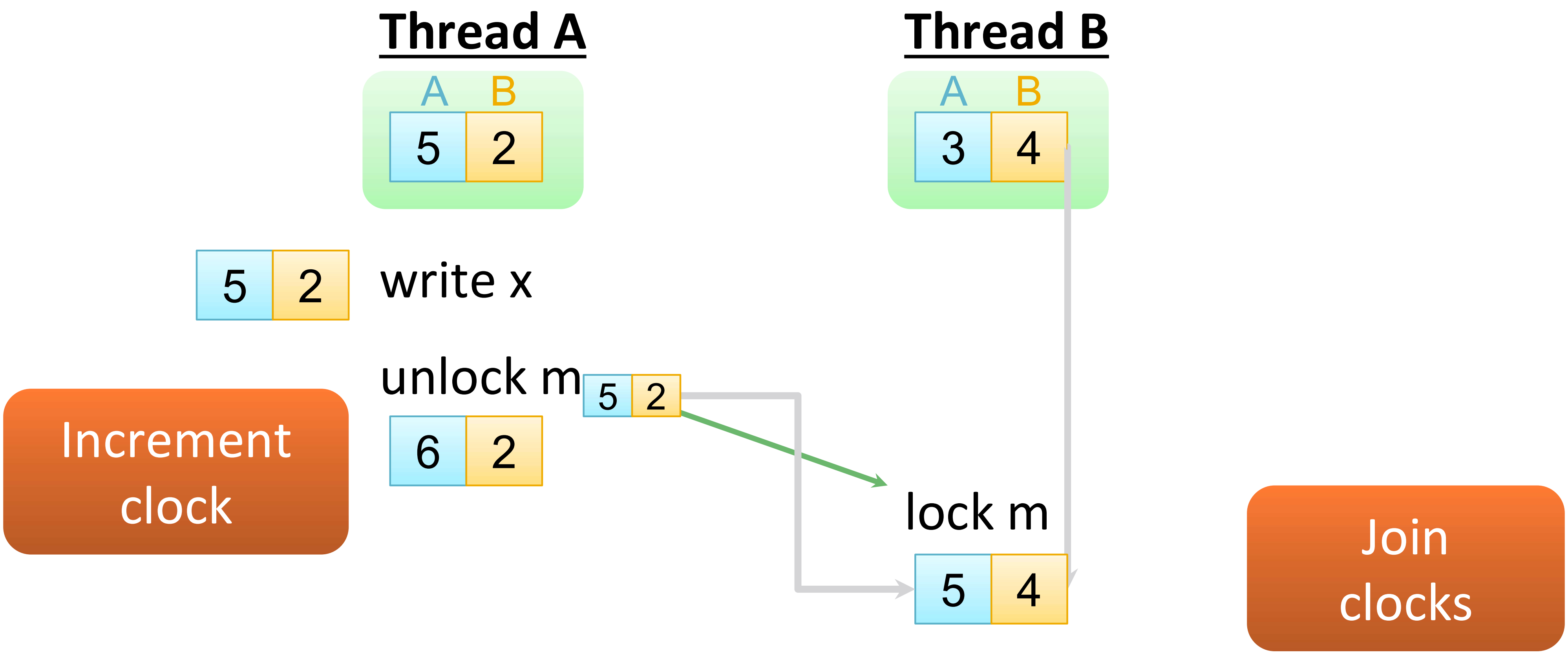
Vector Clock-Based Race Detection



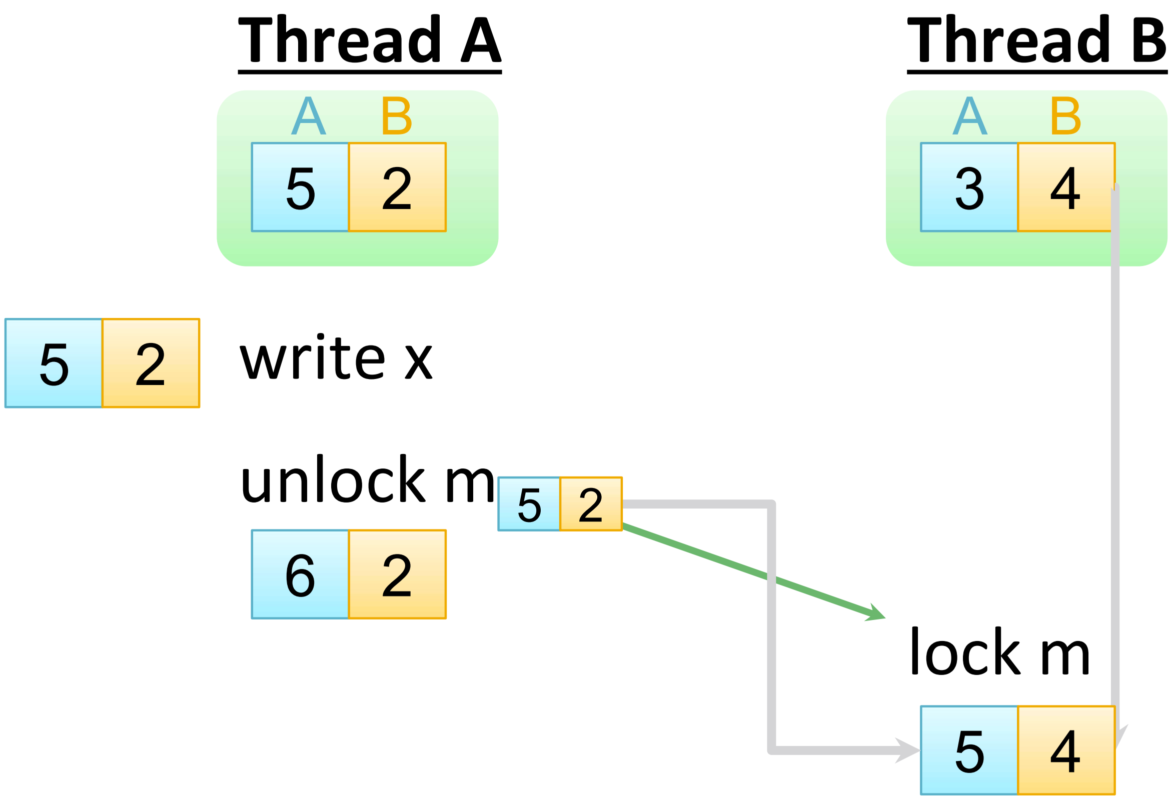
Vector Clock-Based Race Detection



Vector Clock-Based Race Detection



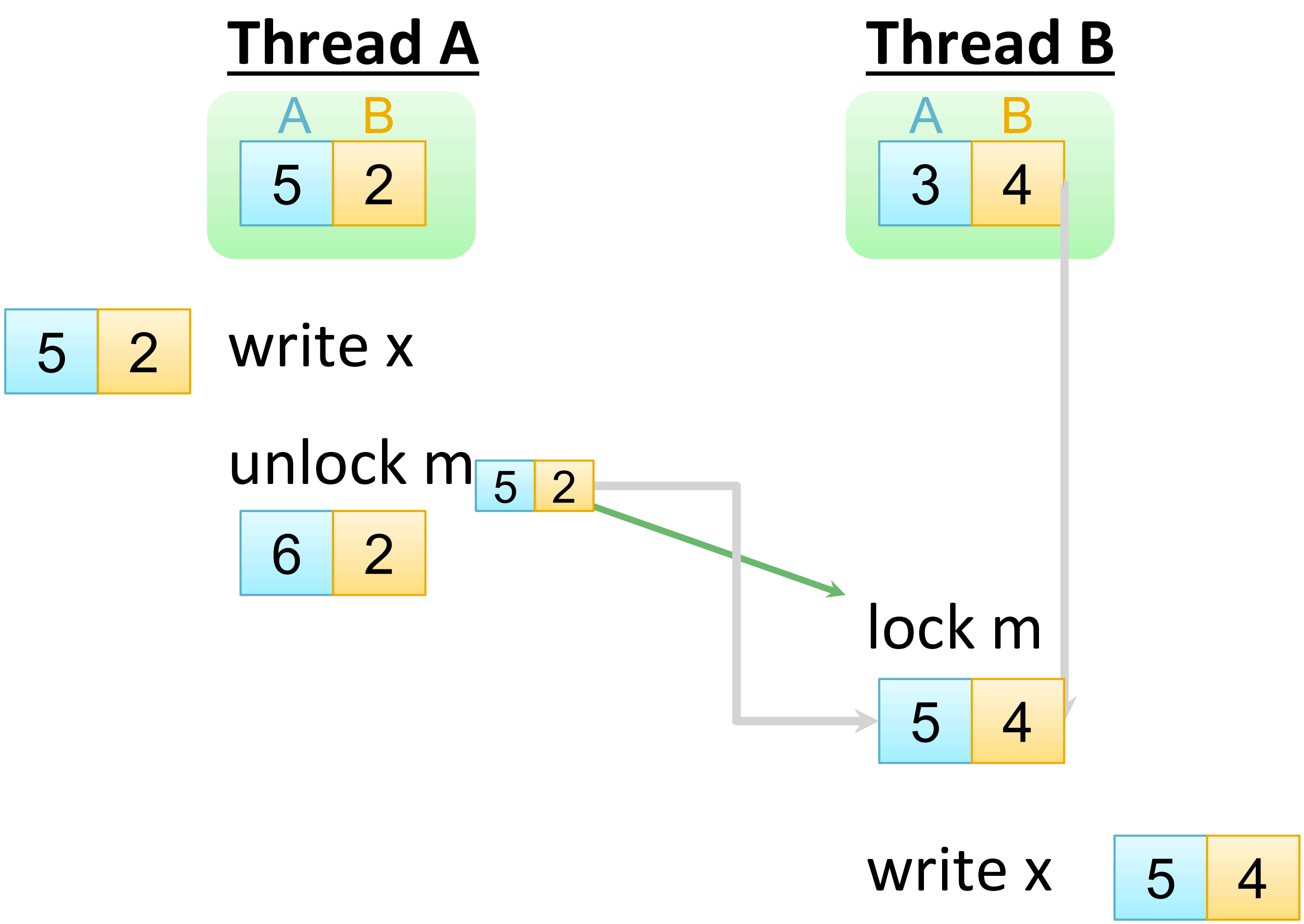
Vector Clock-Based Race Detection



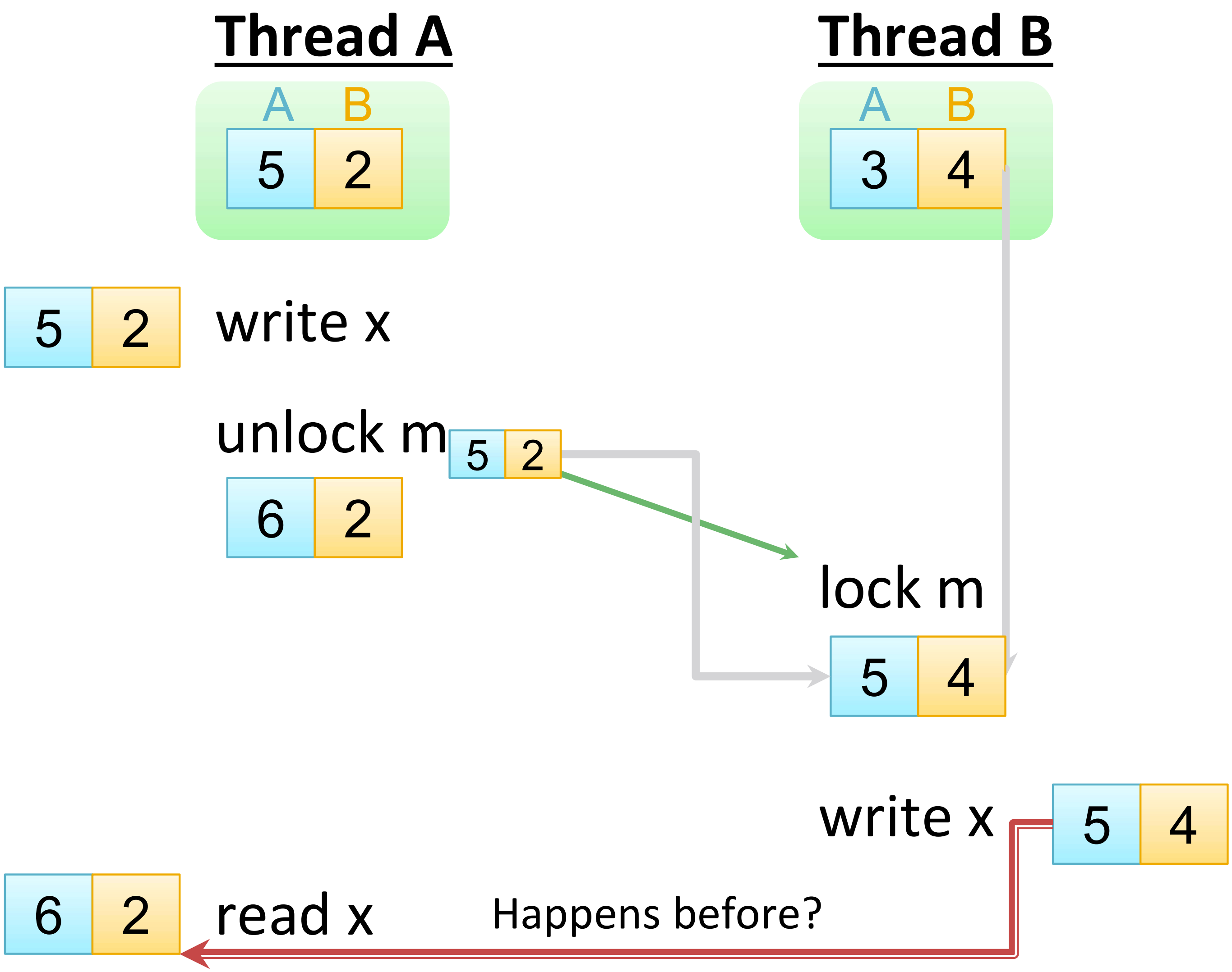
O(n) time

n = # of threads

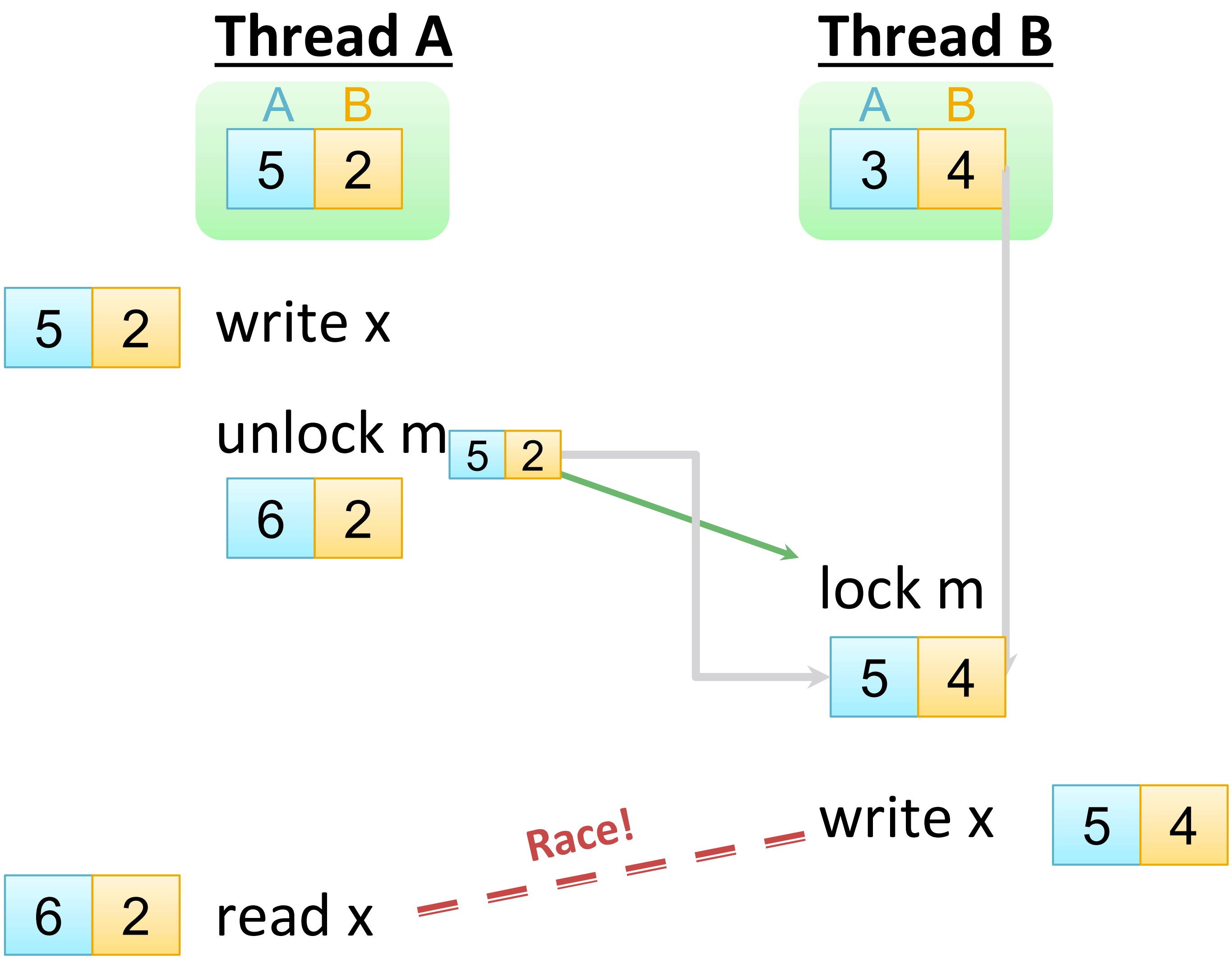
Vector Clock-Based Race Detection



Vector Clock-Based Race Detection



Vector Clock-Based Race Detection



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 = # 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 = # 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

A	B
5	2

write x

unlock m

read x

Thread B

A	B
3	4

lock m

write x

Vector Clock-Based Race Detection

Thread A

A	B
5	2

write x 5@A

unlock m

read x

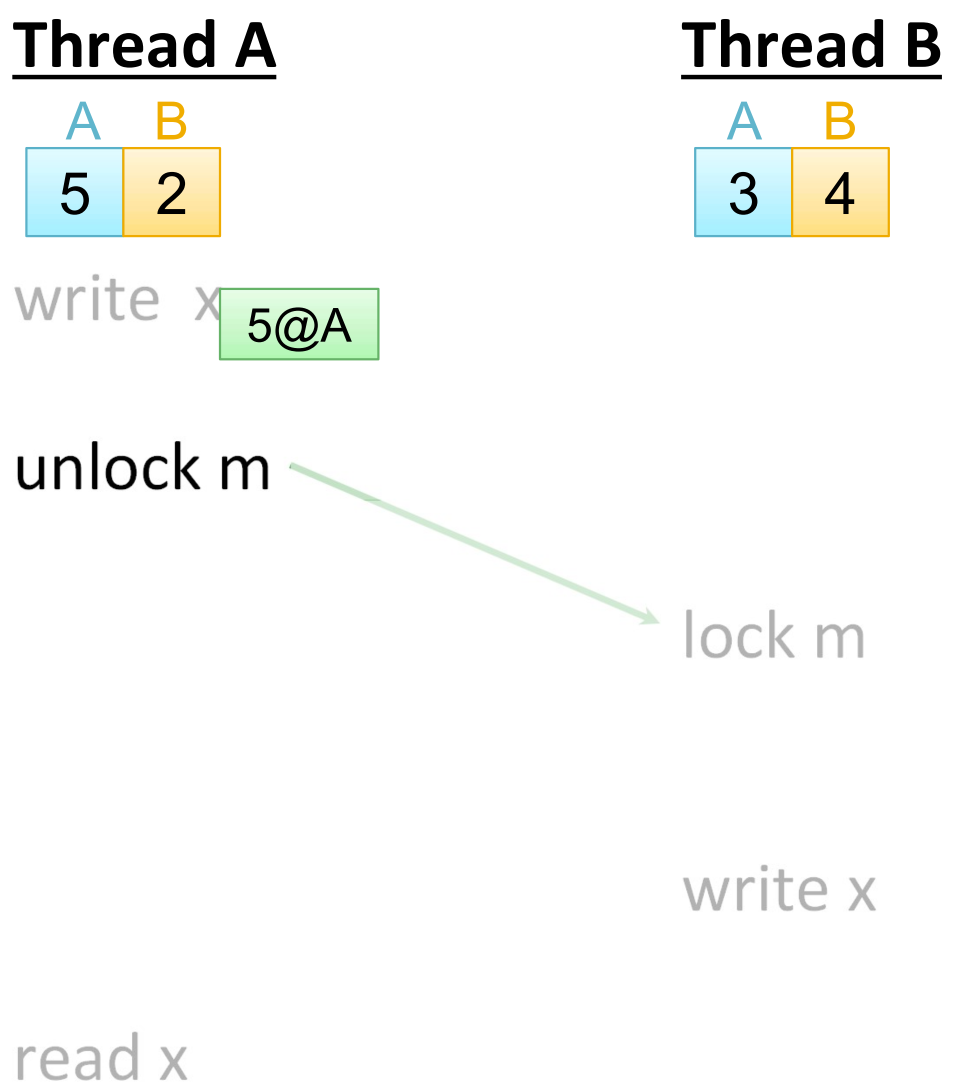
Thread B

A	B
3	4

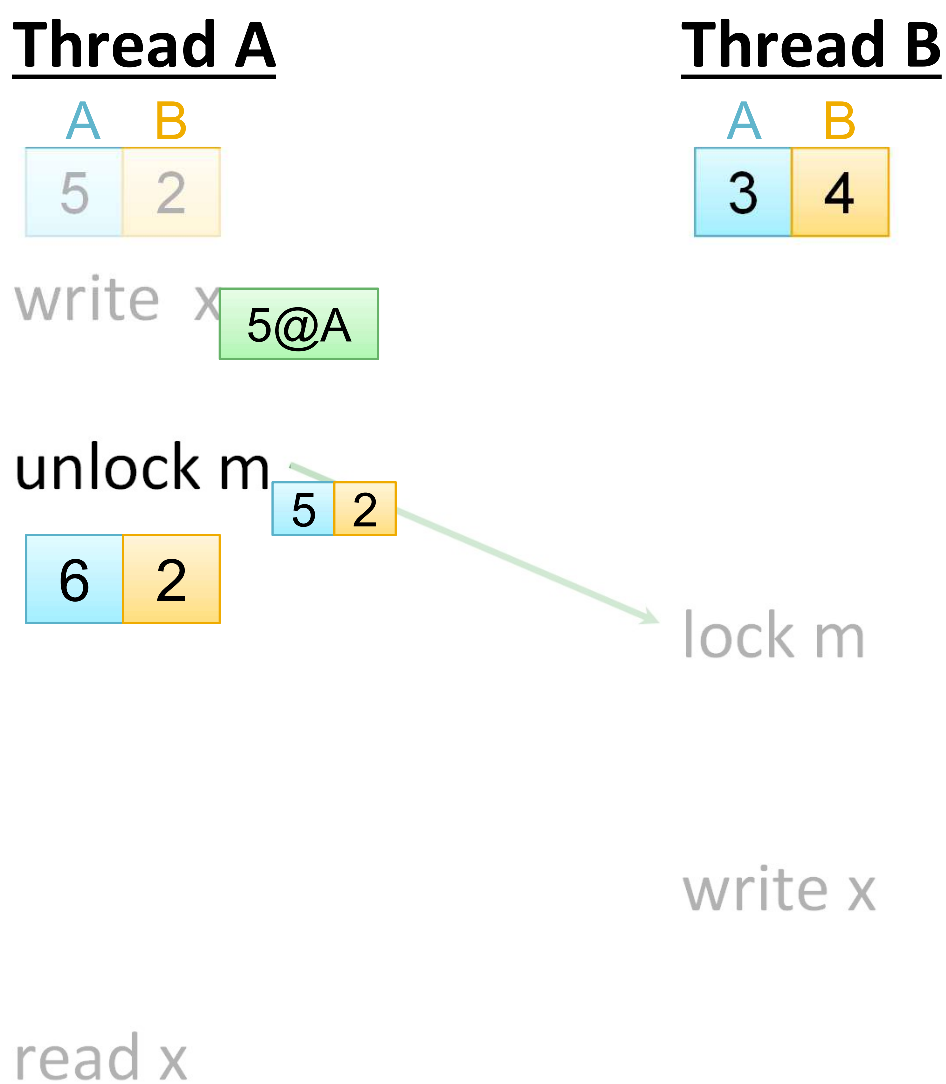
lock m

write x

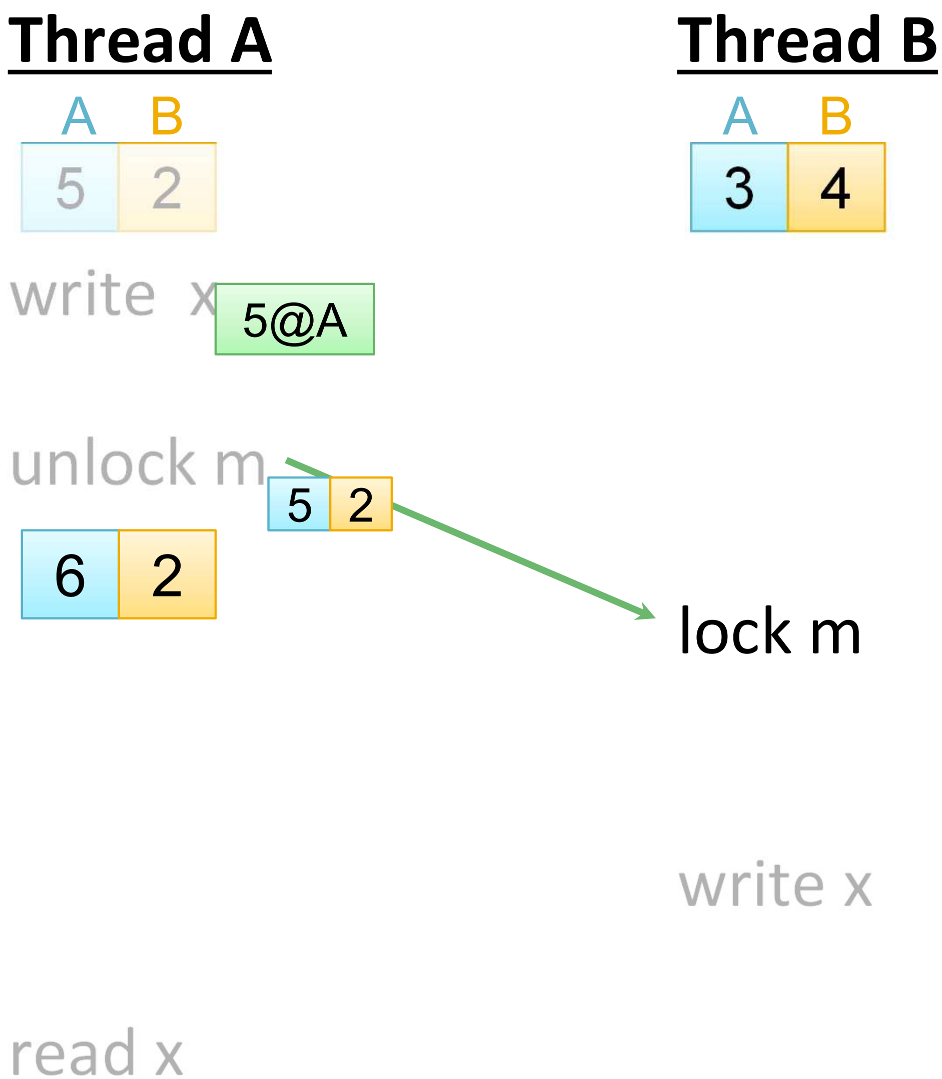
Vector Clock-Based Race Detection



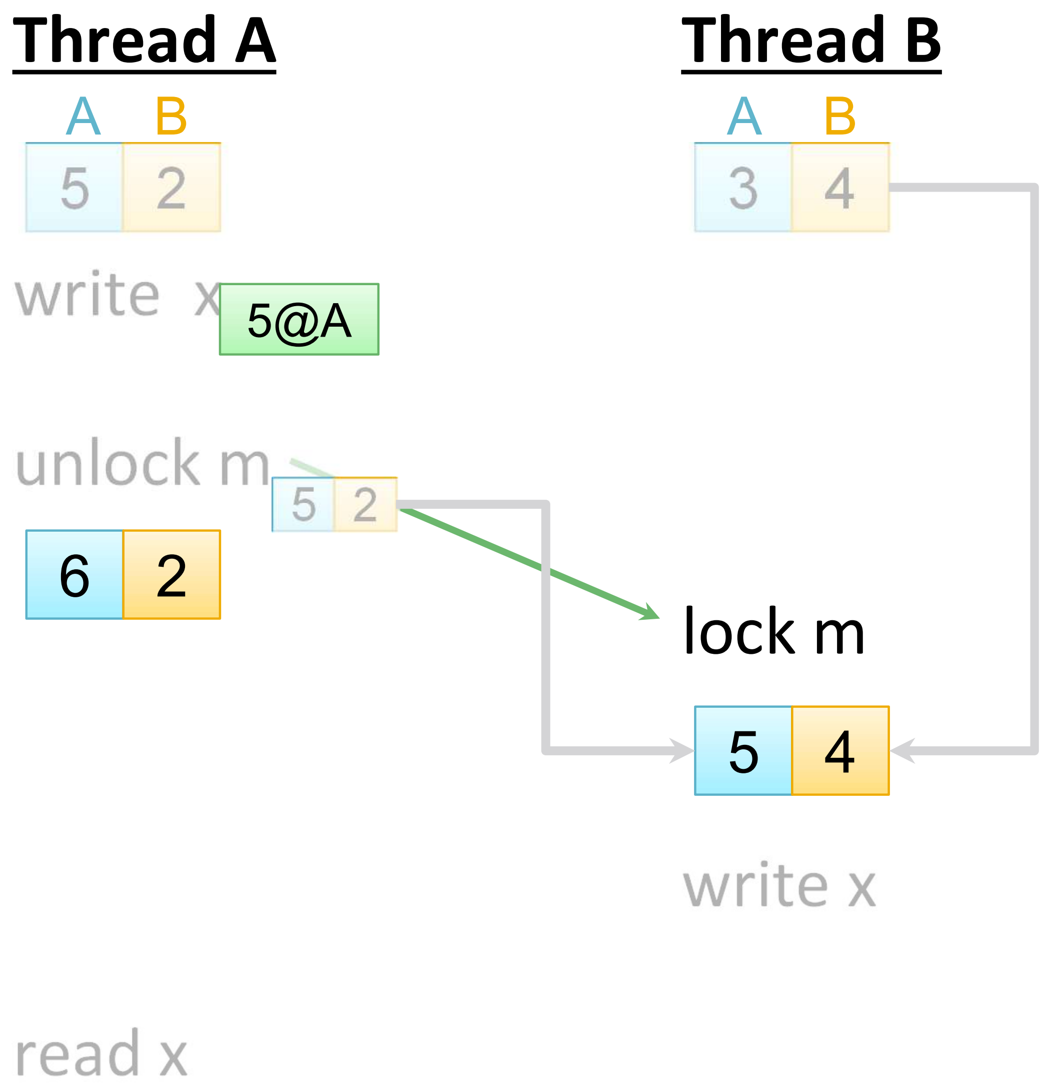
Vector Clock-Based Race Detection



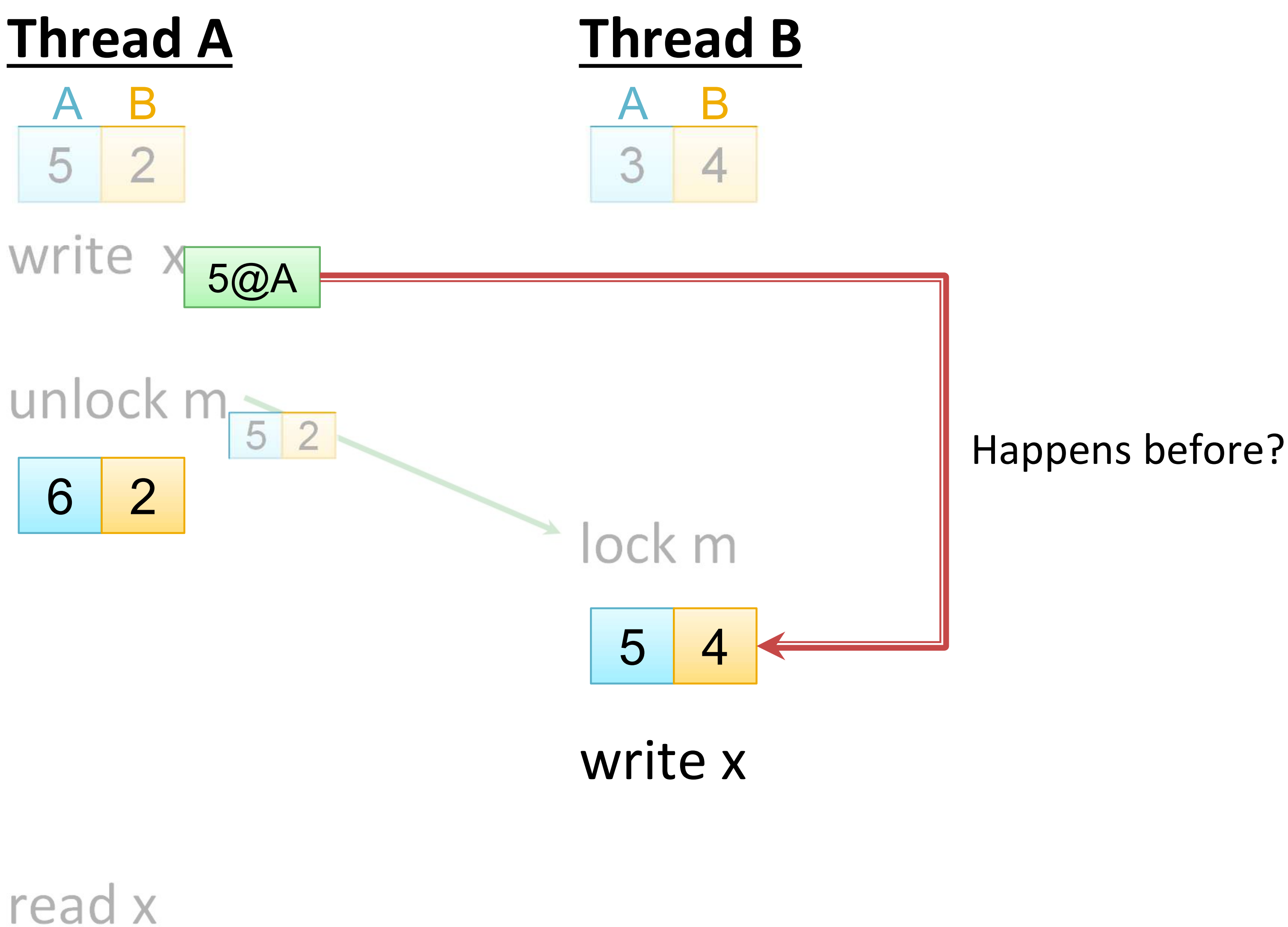
Vector Clock-Based Race Detection



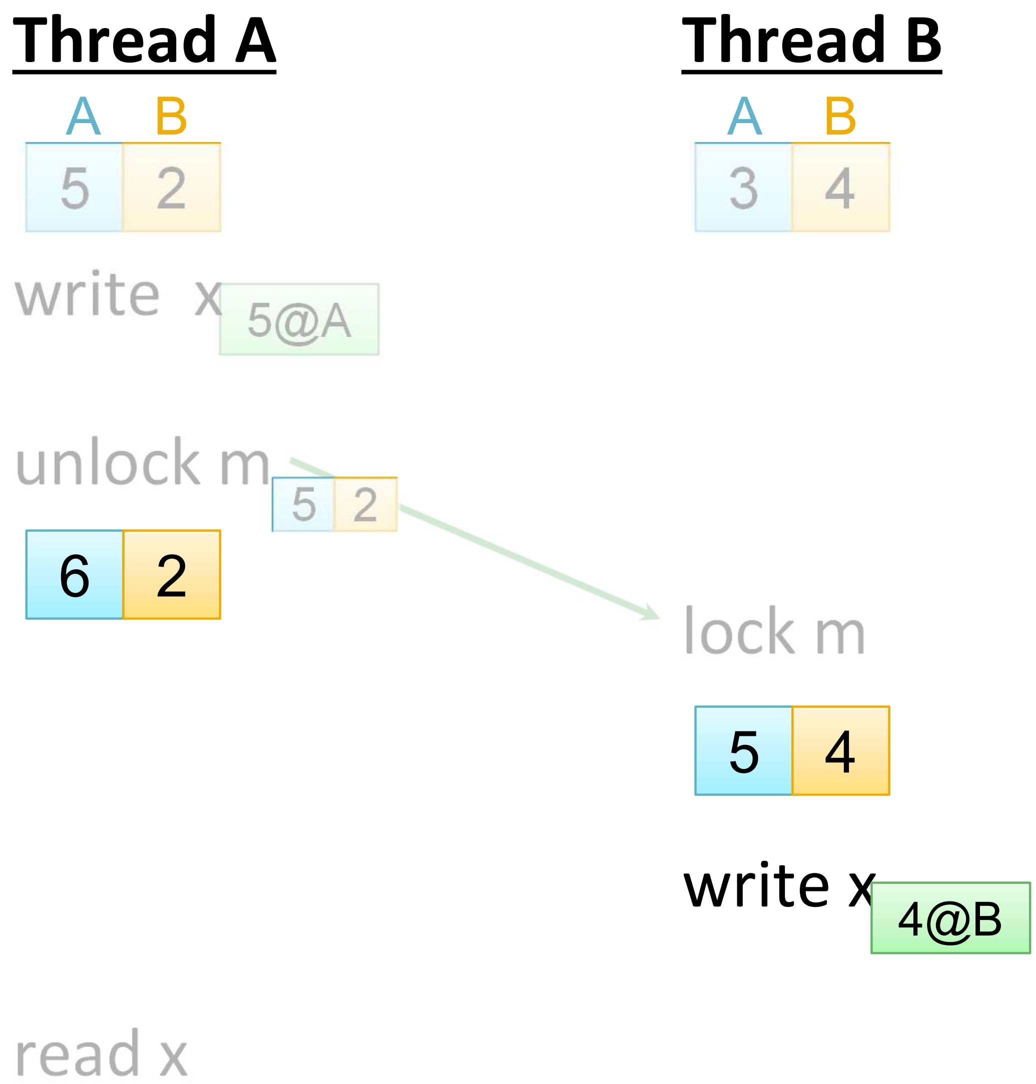
Vector Clock-Based Race Detection



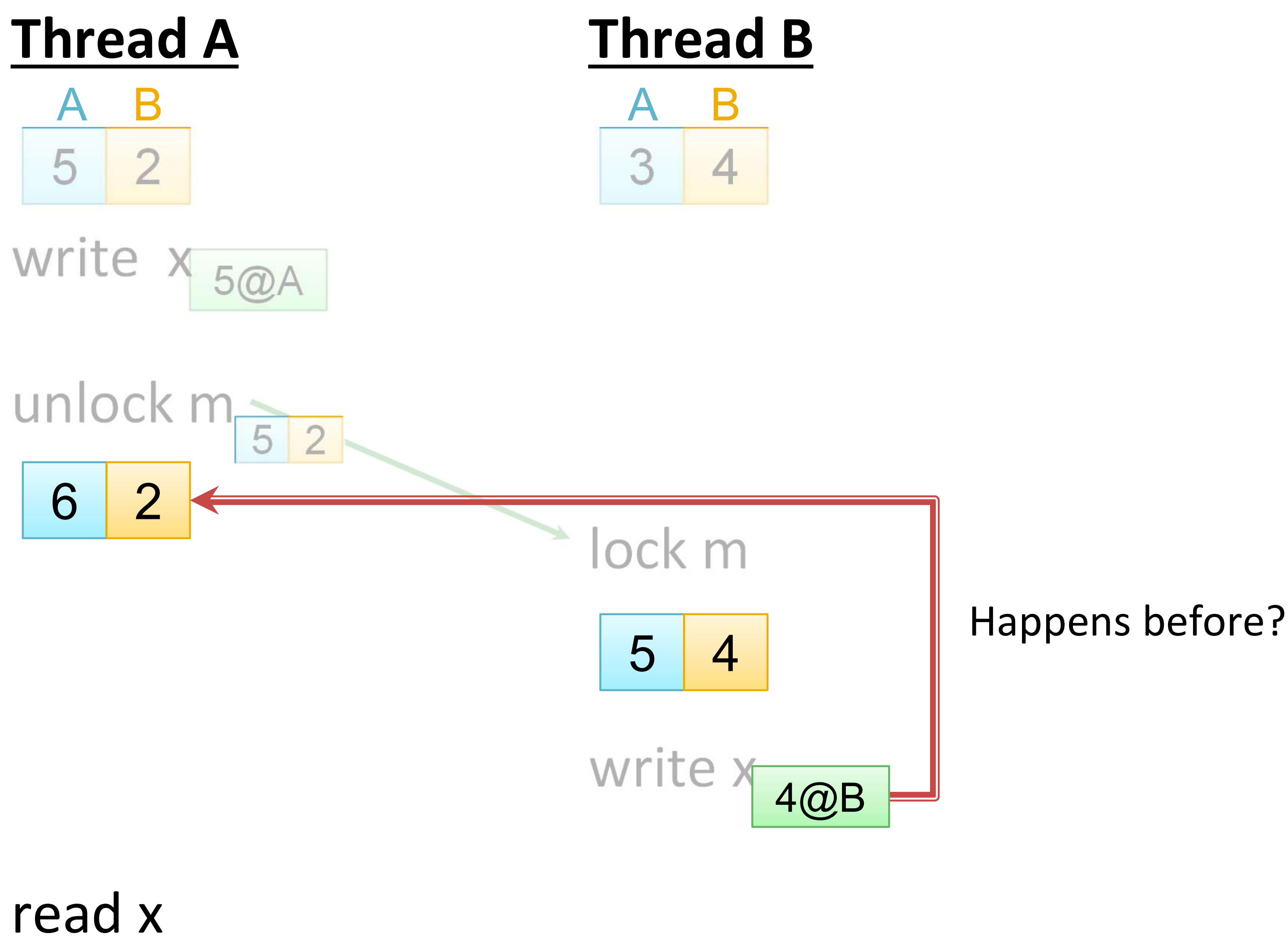
Vector Clock-Based Race Detection



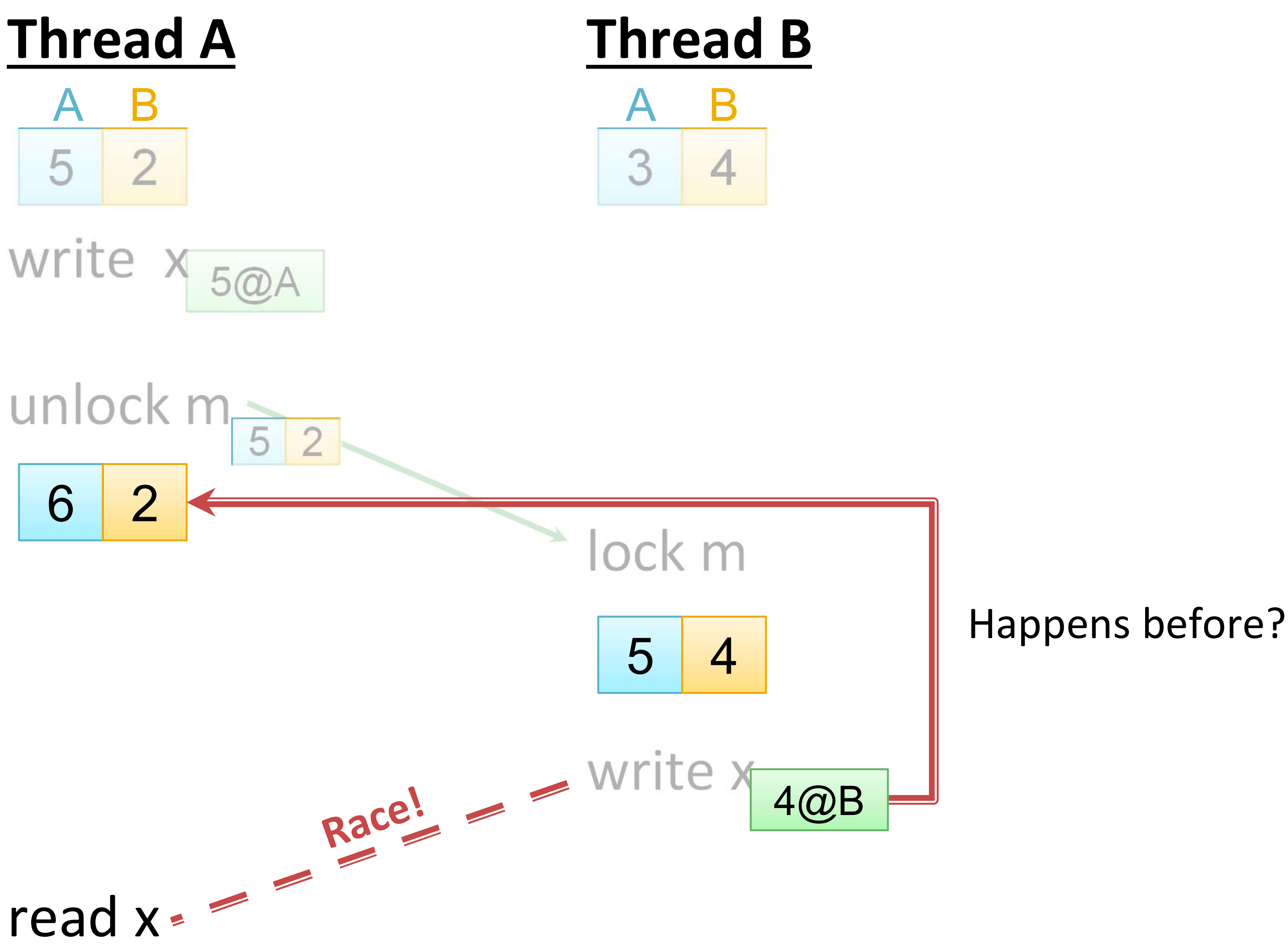
Vector Clock-Based Race Detection



Vector Clock-Based Race Detection



Vector Clock-Based Race Detection



Collision analysis

Basic idea: Make two conflicting accesses happen at the same time

(1) Pause one thread at memory access to x
(2) Catch other threads that access x (in a conflicting way) in the meantime

Can be automatic (with instrumentation or hardware) or manual (programmer adds code)

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

Thread A		**Lockset_A**
lock m	← – – – →	L = { }
write x	← – – – →	L = {m}
lock n		
write y	← – – – →	L = {m, n}
unlock n		
unlock m	← – – – →	L = {m}
read x	← – – – →	L = { }

Lockset Algorithms

- Two accesses from different threads with non-intersecting locksets form a data race
- Can detect more data races than vector clocks
  - A property referred to as **coverage**

Is there a data race on variable y? Observed interleaving

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

Happens before

Is there a data race on variable y?
Observed interleaving

Thread A

Thread B

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

Happens before?

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

Data race with vector clock

Data race with lockset

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

No data race with vector clock

Alternate Interleaving

Thread A		**Thread B**
y = y + 1		
lock m		
v = v + 1		
unlock m	Happens before	
		lock m
		v = v + 1
		unlock m
		y = y + 1

Data race with lockset

Data Race on y with Lockset

Thread A	Thread B	Lockset
y = y + 1		$L_y = \{\ \}$
lock m		
v = v + 1		$L_v = \{m\}$
unlock m		
	lock m	
	v = v + 1	$L_v = \{m\}$
	unlock m	
	y = y + 1	$L_y = \{\ \}$

Lockset Algorithms are Imprecise

```
volatile boolean flag = false;
Object x;
```

Thread A

```
x = new Object();
flag = true;
```

Happens before

Thread B

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

Lockset Algorithms are Imprecise

```
volatile boolean flag = false;
Object x;
```

Thread A

```
x = new Object();
flag = true;
```

Happens before

Thread B

```
while (!flag);

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

B is blocked

Vector Clock 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;
```

Atomicity violation?

```
int x = 1;
```

T1:

```
t = x;


t = t + 1;

x = t;
```

T2:

```
t = x;

t = t * 2;


x = t;
```

Atomicity violation?

```
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;
}
```

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) { ... }
}
```

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);
      }
    }
  }
}
```

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?

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
→ only a 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) {
      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);
  }
}
```

This is (incorrect) 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

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


 comments.add(s);
}
```

```
addComment(String s) {







 if (comments == null) {
 }
 comments.add(s);
}
```

Another example: double-checked locking

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

 comments.add(s);
}
```

```
addComment(String s) {

 if (comments == null) {
 }
 comments.add(s);
}
```

Another example: double-checked locking

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


 comments.add(s);
}
```

```
addComment(String s) {








 if (comments == null) {
 }
 comments.add(s);
}
```

Another example: double-checked locking

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

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

This is correct double-checked locking

```
class Movie {
  volatile Vector<String> comments;

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