

# Concurrency Errors (1)

lecture 22 (2021-05-31)

**Master in Computer Science and Engineering**

— Concurrency and Parallelism / 2020-21 —

João Lourenço <joao.lourenco@fct.unl.pt>

# Agenda

---

- Concurrency Errors
  - Detection of data races
  - Detection of high-level data races and stale value errors
  - Detection of deadlocks
- Reading list:
  - TBD

# Concurrency Errors

---

## Data Race Detection

# Overview

---

- Static program analysis
- Dynamic program analysis
  - Lock-set algorithm
  - Happens-Before
  - Noise-Injection

# Static Data Race Detection

- Advantages:
  - Reason about all inputs/interleavings
  - No run-time overhead
  - Adapt well-understood static-analysis techniques
  - Possibly with annotations to document concurrency invariants
- Example Tools:
  - RCC/Java                      type-based
  - ESC/Java                      "functional verification"  
   (theorem proving-based)

# Static Data Race Detection

- Advantages:
  - Reason about all inputs/interleavings
  - No run-time overhead
  - Adapt well-understood static-analysis techniques
  - Possibly with annotations to document concurrency invariants
- Disadvantages of static approach:
  - Tools produce “false positives” and/or “false negatives”
  - May be slow, require programmer annotations
  - May be hard to interpret results
  - May not scale to large or complex programs

# Dynamic Data Race Detection

- Advantages

- Soundness
  - Every actual data race is reported
- Completeness
  - All reported warnings are actually races (avoid “false positives”)

- Disadvantages

- Run-time overhead (5-20x for best tools)
- Memory overhead for analysis state
- Reasons only about observed executions
  - sensitive to test coverage
  - (some generalization possible...)

# Approaches

---

- Happens-Before
- Lock-set algorithm
  - Learns which shared memory locations are protected by which locks
  - Issues warning if finds no lock protects a shared memory location
- (...)



# Concurrency Errors

---

Dynamic Data Race Detection Using  
Happens-before [Lamport '78]

# Lock Definition

---

- **Lock**: a synchronization object that is either available, or owned (by a thread)
  - Operations: **lock(mu)** and **unlock(mu)**
    - *(We are assuming no explicit initialize operation)*
  - A lock can only be unlocked by its current owner
  - The **lock()** operation is blocking if the lock is owned by another thread

# The Happens-before Relation

- *happens-before* defines a partial order for events in a set of concurrent threads
  - In a single thread, *happens-before* reflects the temporal order of event occurrence
  - Between threads, **A** happens before **B** if A is an unlock access in one thread, and **B** is a lock access in a **different** thread (*assuming the threads obey the semantics of the lock , i.e., can't have two successive locks, or two successive unlocks, or a lock in one thread and an unlock in a different thread*)

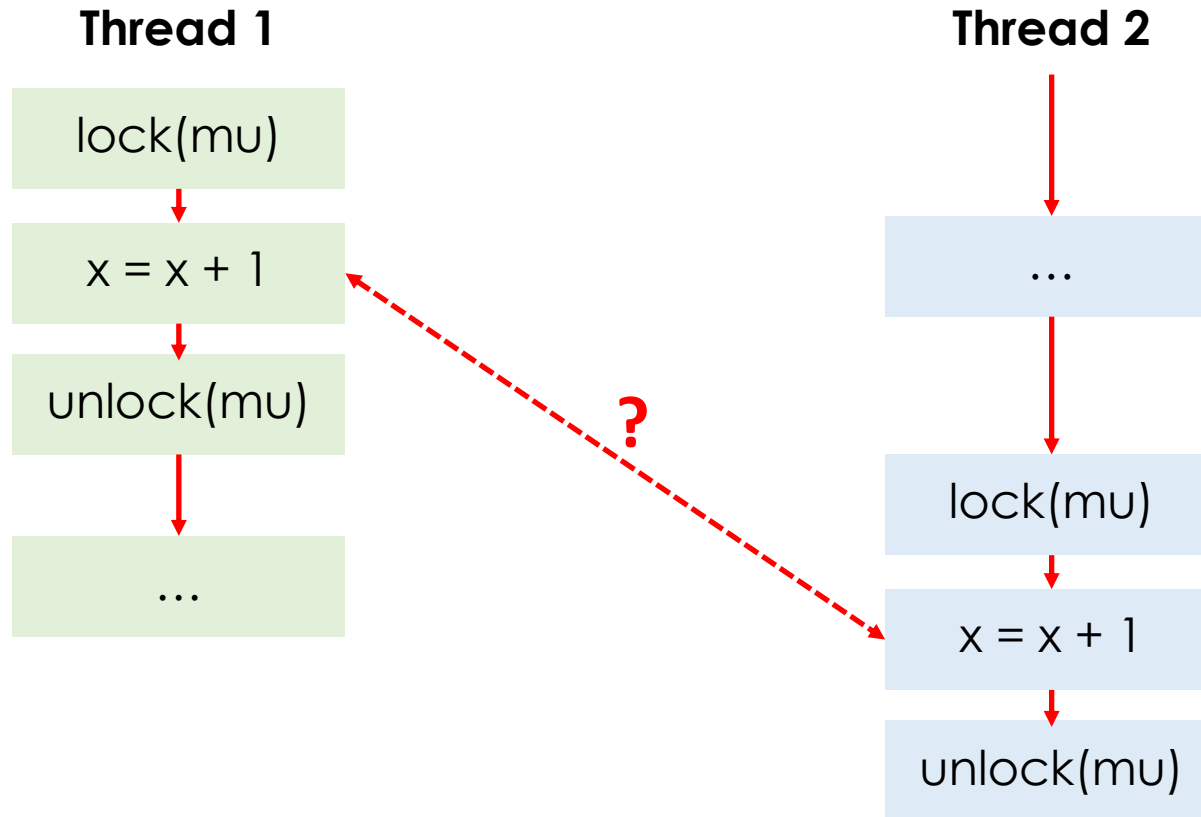
# The Happens-before Relation

- Let **event a** be in thread 1 and **event b** be in thread 2

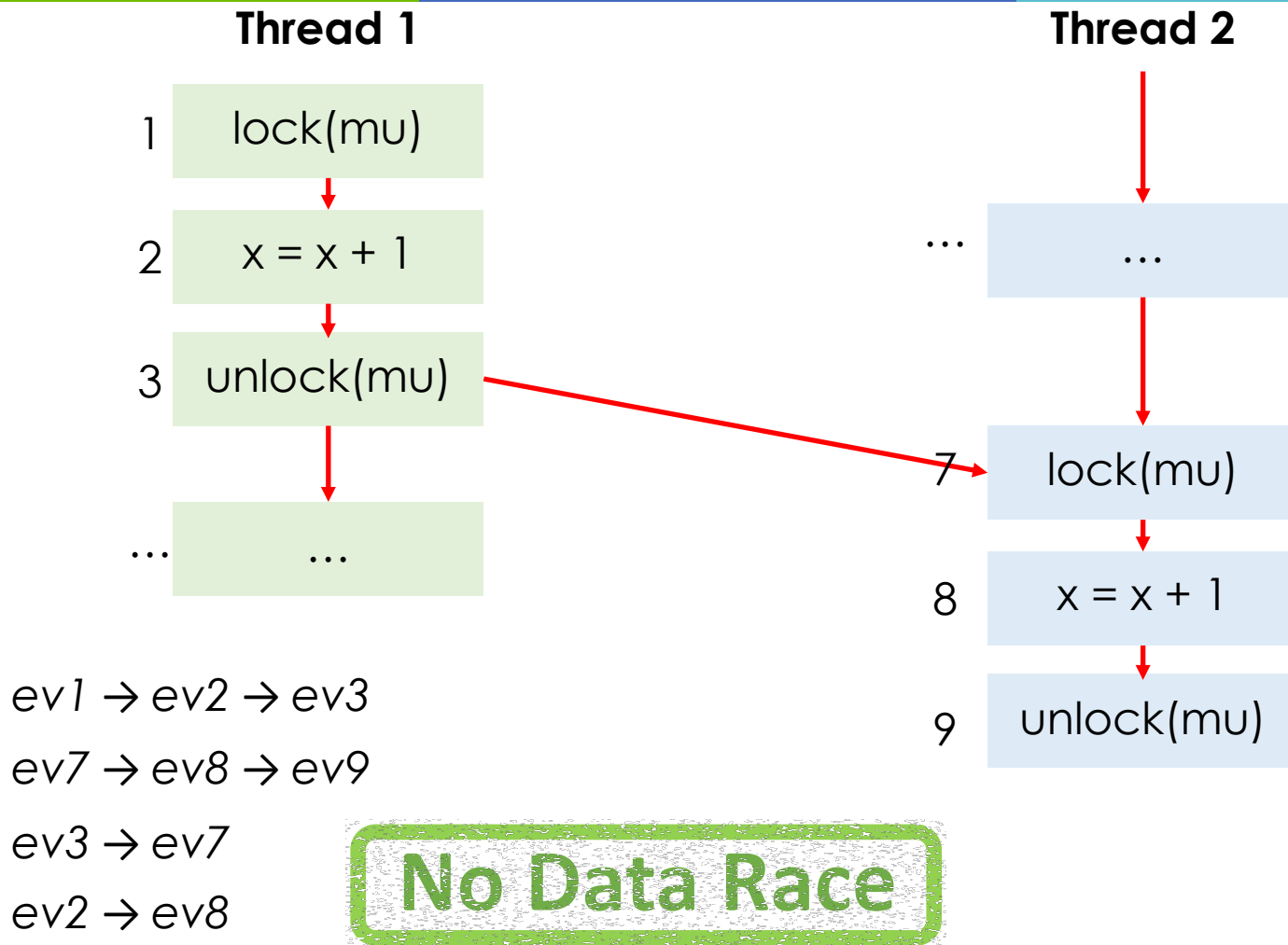
*If  $a = \text{unlock}(\mu)$  and  $b = \text{lock}(\mu)$  then  
 $a \rightarrow b$  (a happens-before b)*

Data races between threads are **possible** if accesses to shared variables are not ordered by the *happens-before* relation

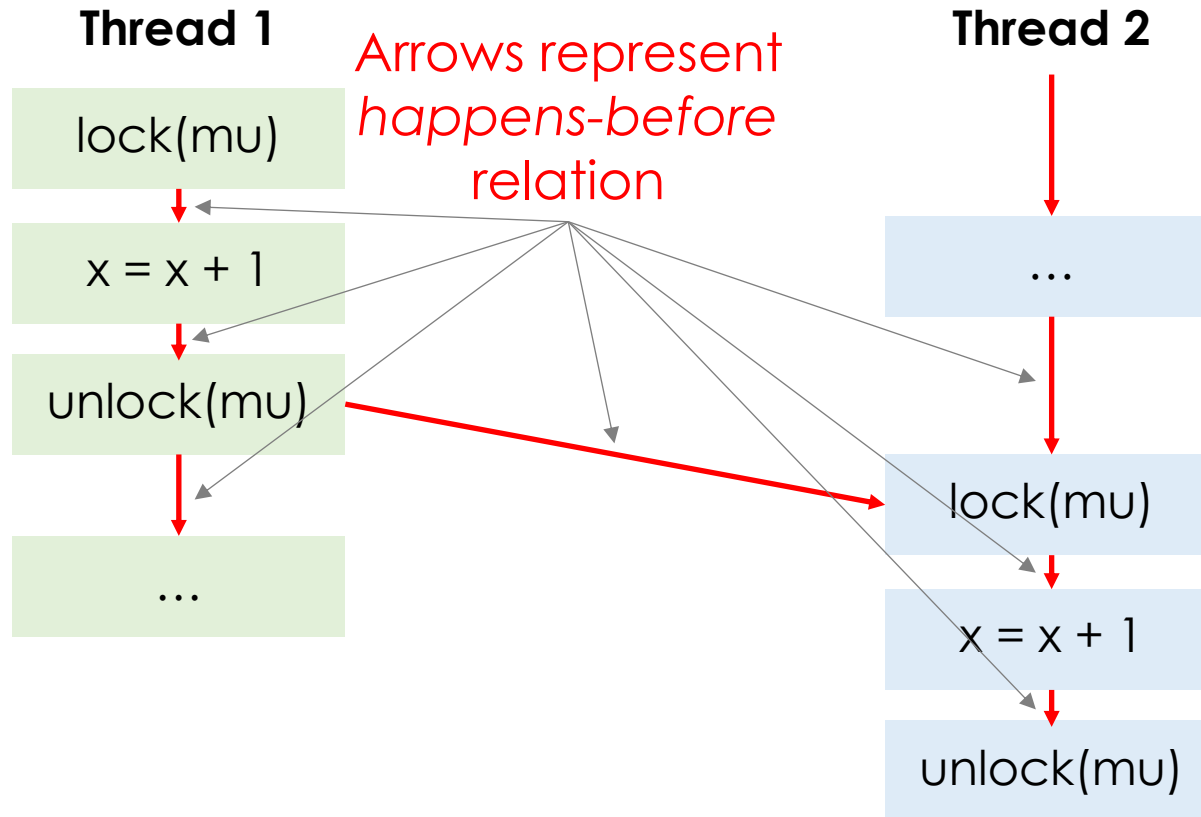
# Example 1



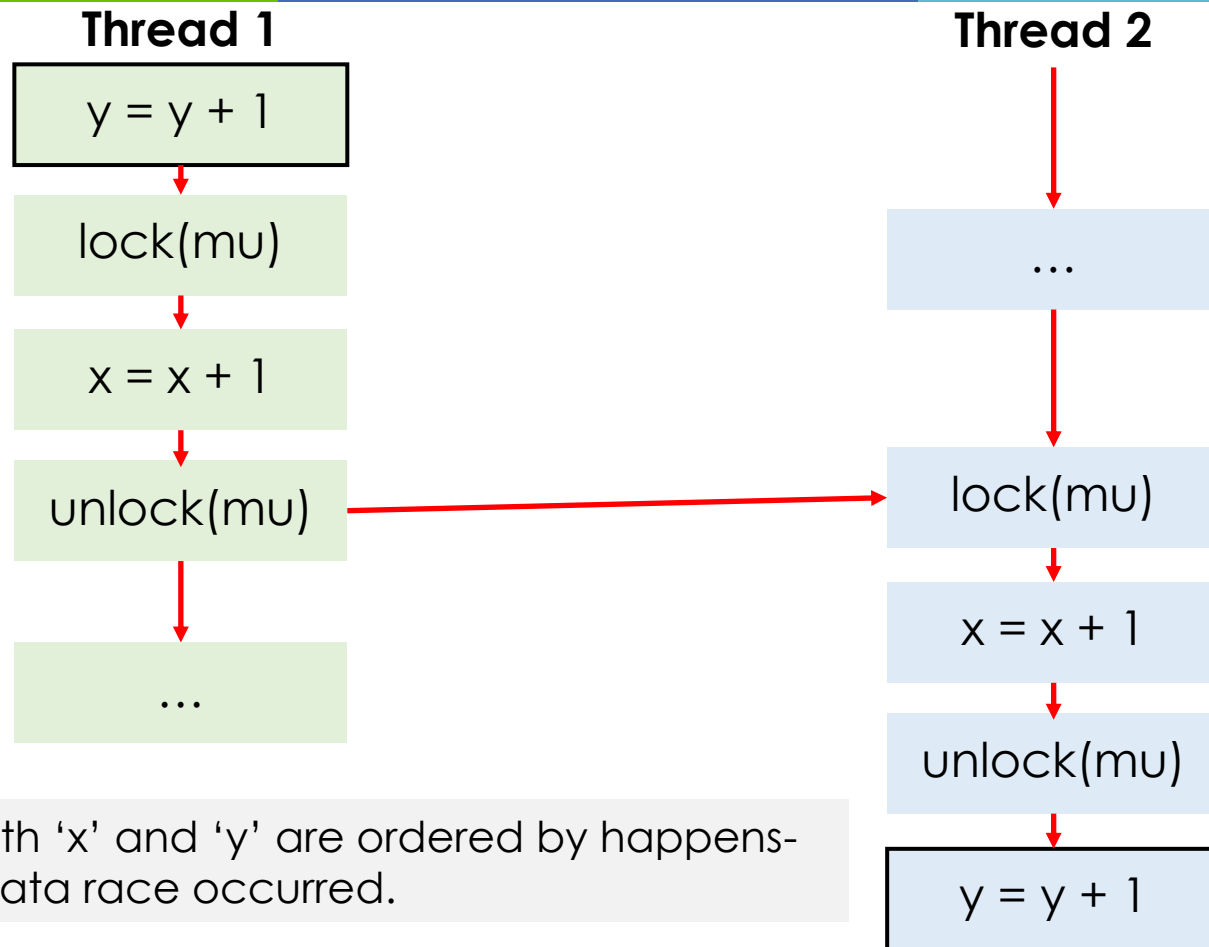
# Example 1



# Example 1



# Example 2

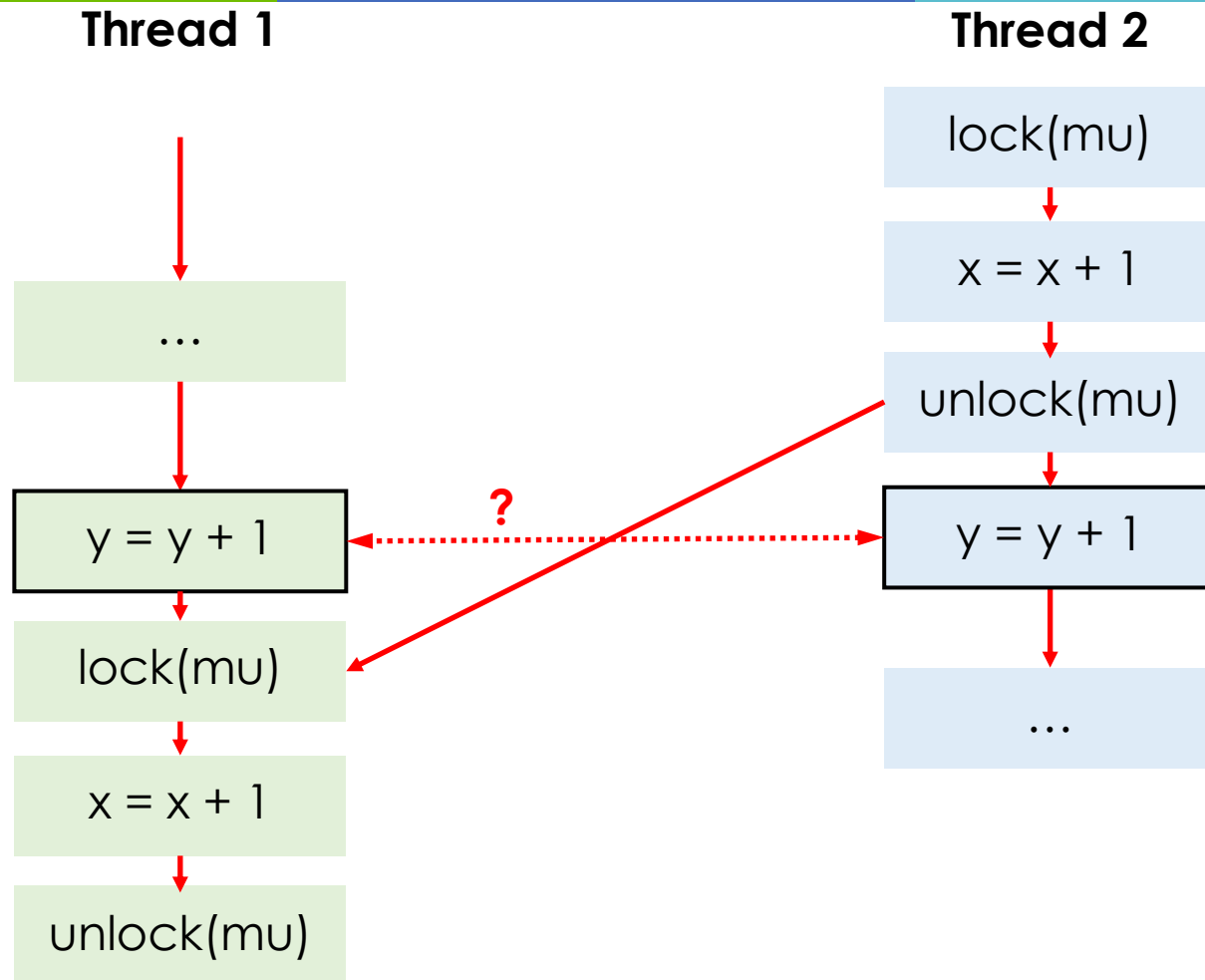


Accesses to both 'x' and 'y' are ordered by happens-before, so no data race occurred.

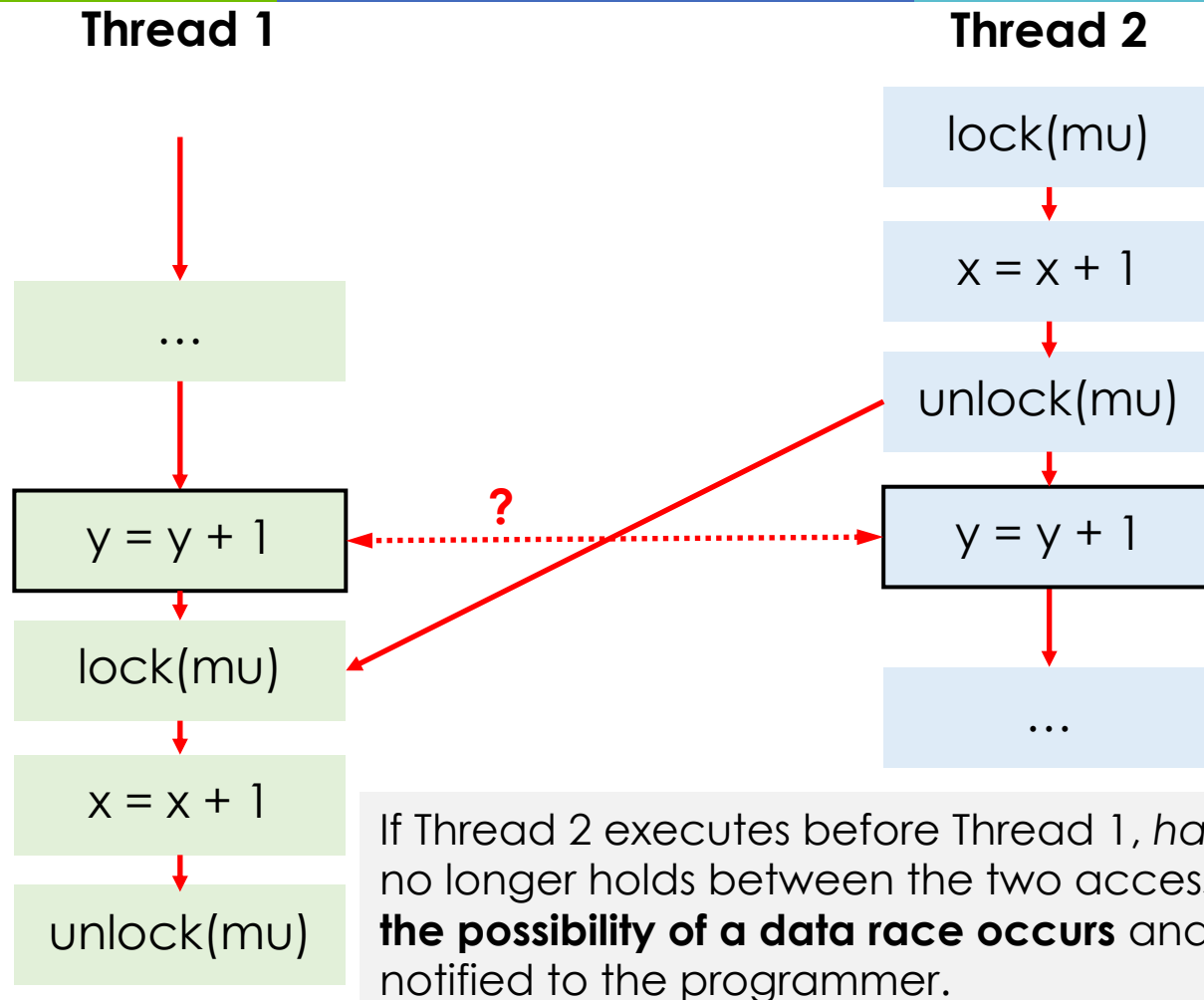
But ... a different execution ordering could get different results?! Happens-before only detects data races if the incorrect order shows up in the execution trace.



# Example 3



# Example 3



# Concurrency Errors

The Lock-Set Algorithm — Eraser [Savage et.al. '97]

# Approaches

- Checks a sufficient condition for data-race freedom
- Consistent locking discipline
  - Every data structure is protected by a single lock
  - All accesses to the data structure are made while holding the lock

## Thread 1

```
void Bank::Deposit(int a) {  
  
    int t = bal;  
    bal = t + a;  
  
}
```

## Thread 2

```
void Bank::Withdraw(int a) {  
  
    int t = bal;  
    bal = t - a;  
  
}
```

# Approaches

- Checks a sufficient condition for data-race freedom
- Consistent locking discipline
  - Every data structure is protected by a single lock
  - All accesses to the data structure are made while holding the lock

## Thread 1

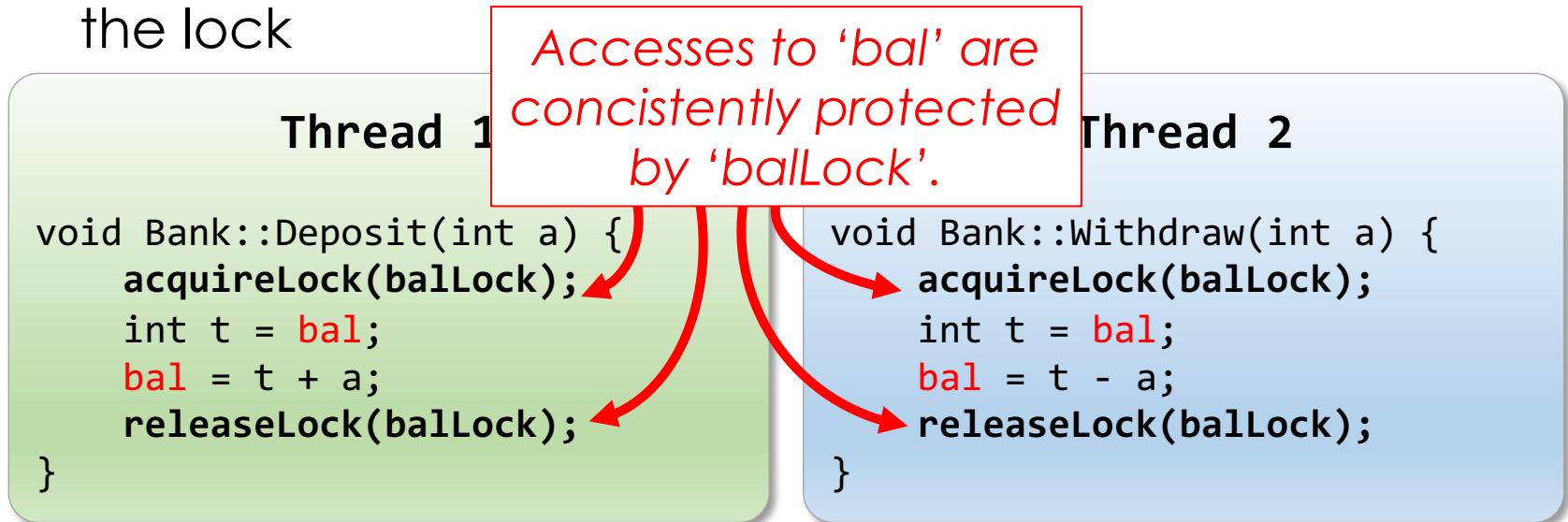
```
void Bank::Deposit(int a) {  
    acquireLock(balLock);  
    int t = bal;  
    bal = t + a;  
    releaseLock(balLock);  
}
```

## Thread 2

```
void Bank::Withdraw(int a) {  
    acquireLock(balLock);  
    int t = bal;  
    bal = t - a;  
    releaseLock(balLock);  
}
```

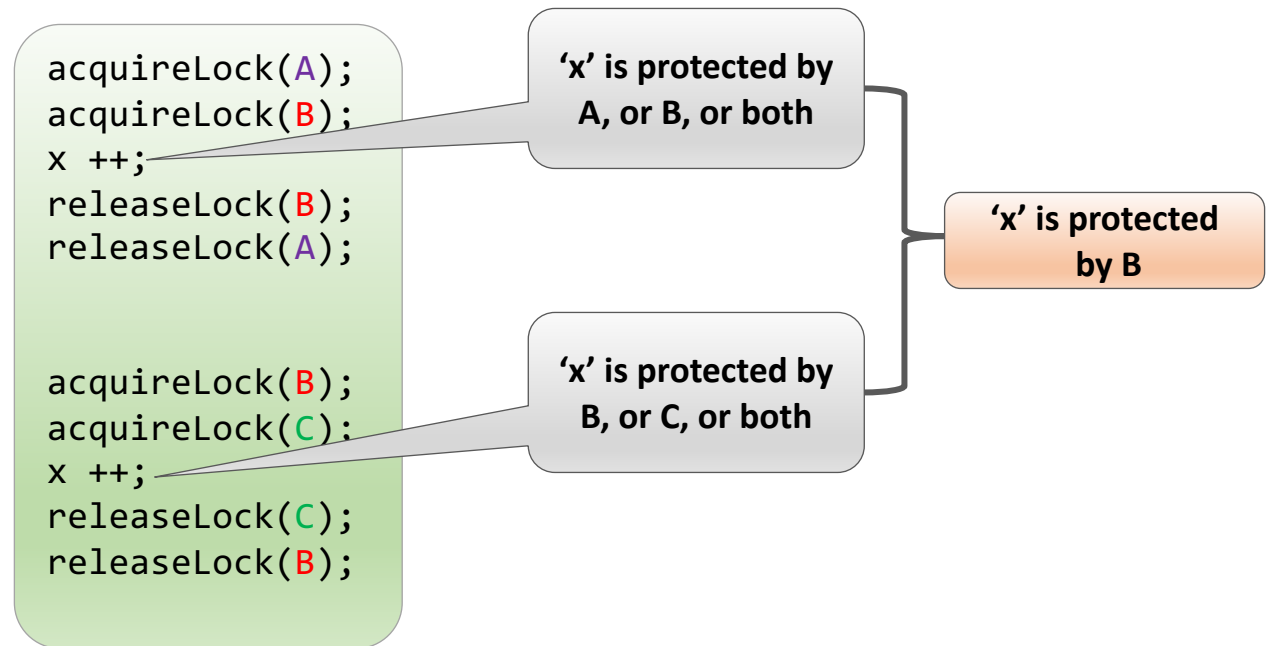
# Approach

- Checks a sufficient condition for data-race freedom
- Consistent locking discipline
  - Every data structure is protected by a single lock
  - All accesses to the data structure are made while holding the lock



# Approach

- How to know which locks protect each memory location?
  - Ask the programmer? Cumbersome!
  - Infer from the program code? Is it effective?



# The Lock-Set Algorithm

- Two data structures:
  - $\text{LocksHeld}(t)$  = set of locks held currently by thread  $t$ 
    - Initially set to Empty
  - $\text{LockSet}(x)$  = set of locks that could potentially be protecting  $x$ 
    - Initially set to the universal set
- When thread ' $t$ ' acquires lock ' $l$ '
  - $\text{LocksHeld}(t) = \text{LocksHeld}(t) \cup \{l\}$
- When thread ' $t$ ' releases lock ' $l$ '
  - $\text{LocksHeld}(t) = \text{LocksHeld}(t) \setminus \{l\}$
- When thread ' $t$ ' accesses location ' $x$ '
  - $\text{LockSet}(x) = \text{LockSet}(x) \cap \text{LocksHeld}(t)$
- “Data race” warning if  $\text{LockSet}(x)$  becomes empty



# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)		
lock(m2)		
v = v + 1		
unlock(m2)		
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
<b>lock (m1)</b> → <b>U</b> → <b>lock (m1)</b>	{m1}	{m1, m2}
lock(m2)		
v = v + 1		
unlock(m2)		
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
<b>lock(m2)</b>	<b>{m1, m2}</b>	{m1, m2}
v = v + 1		
unlock(m2)		
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
<b>v = v + 1</b>	{m1, m2} → ∩	{m1, m2}
unlock(m2)		
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
<b>unlock(m2)</b>	<b>{m1}</b>	{m1, m2}
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
<b>v = v + 2</b>	{m1} $\longrightarrow$ $\cap$	<b>{m1}</b>
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1}	{m1}
unlock(m1)	{ }	{m1}
lock(m2)		
v = v + 1		
unlock(m2)		



# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1}	{m1}
unlock(m1)	{ }	{m1}
<b>lock(m2)</b> → <b>U</b> →	<b>{m2}</b>	{m1}
v = v + 1		
unlock(m2)		



# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1}	{m1}
unlock(m1)	{ }	{m1}
lock(m2)	{m2}	{m1}
<b>v = v + 1</b>	{m2} → ∩ ← {m1}	{ }
unlock(m2)		

# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1}	{m1}
unlock(m1)	{ }	{m1}
lock(m2)	{m2}	{m1}
v = v + 1		{ } — <b>ALARM</b>
unlock(m2)		

# Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1}	{m1}
unlock(m1)	{ }	{m1}
lock(m2)	{m2}	{m1}
v = v + 1		<b>{ } — ALARM</b>
unlock(m2)	{ }	{ }



# Algorithm Guarantees

---

- No warnings  $\Rightarrow$  no data races on the current execution
  - The program followed consistent locking discipline in this execution
- Warnings does not imply a data race
  - Thread-local initialization or Bad locking discipline

# Algorithm Guarantees

- No warnings  $\Rightarrow$  no data races on the current execution
  - The program followed consistent locking discipline in this execution
- Warnings does not imply a data race
  - Thread-local initialization or **Bad locking discipline**

## Thread 1

```
acquireLock(m1);  
acquireLock(m2);  
x = x + 1;  
releaseLock(m2);  
releaseLock(m1);
```

## Thread 2

```
acquireLock(m2);  
acquireLock(m3);  
x = x + 1;  
releaseLock(m3);  
releaseLock(m2);
```

## Thread 3

```
acquireLock(m1);  
acquireLock(m3);  
x = x + 1;  
releaseLock(m3);  
releaseLock(m1);
```



# Acknowledgments

---

- Some parts of this presentation was based in publicly available slides and PDFs
  - [www.cs.cornell.edu/courses/cs4410/2011su/slides/lecture10.pdf](http://www.cs.cornell.edu/courses/cs4410/2011su/slides/lecture10.pdf)
  - [www.microsoft.com/en-us/research/people/madanm/](http://www.microsoft.com/en-us/research/people/madanm/)
  - [williamstallings.com/OperatingSystems/](http://williamstallings.com/OperatingSystems/)
  - [codex.cs.yale.edu/avi/os-book/OS9/slide-dir/](http://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/)

# The END

---