departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Monitoring Concurrency Errors: Detection of Deadlocks, Atomicity Violations, and Data Races (2)

Concurrency and Parallelism — 2017-18

Master in Computer Science

(Mestrado Integrado em Eng. Informática)

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

Agenda

- Why are we here?
- Concurrency Anomalies
- Assigning Semantics to Concurrent Programs
- Concurrency Errors
  - Detection of data races
  - Detection of high-level data races and stale value errors
  - Detection of deadlocks

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*)

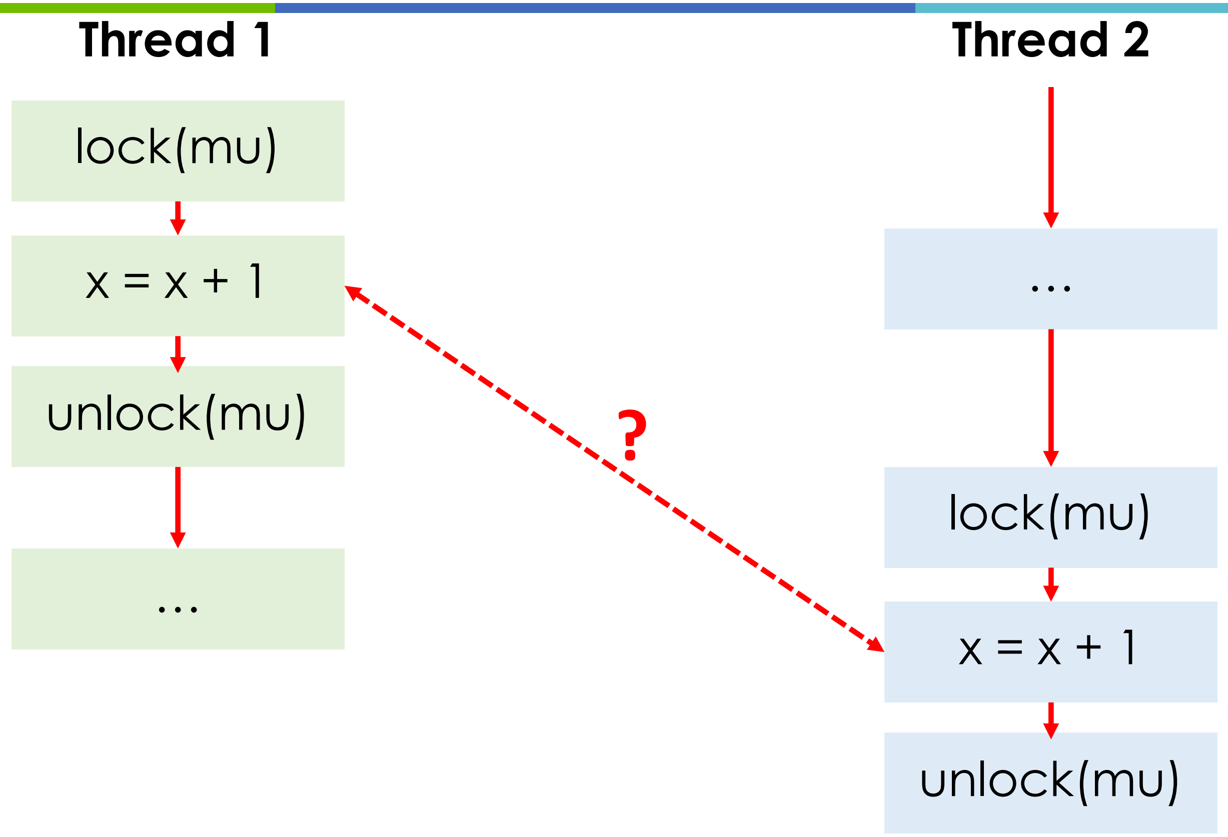
The Happens-before Relation

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

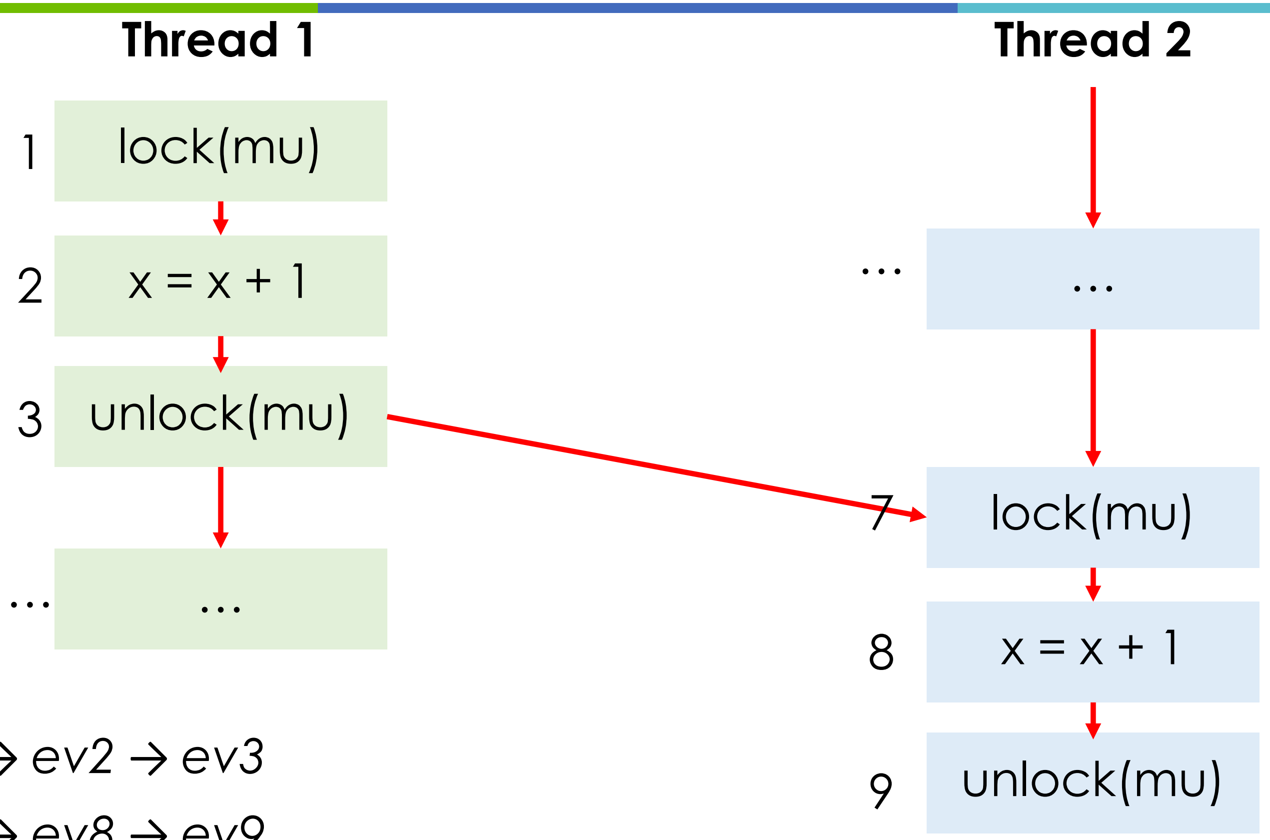
If a = unlock(mu) and b = lock(mu) then
a → b (a happens-before b)

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

Example 1



Example 1



ev1 → ev2 → ev3

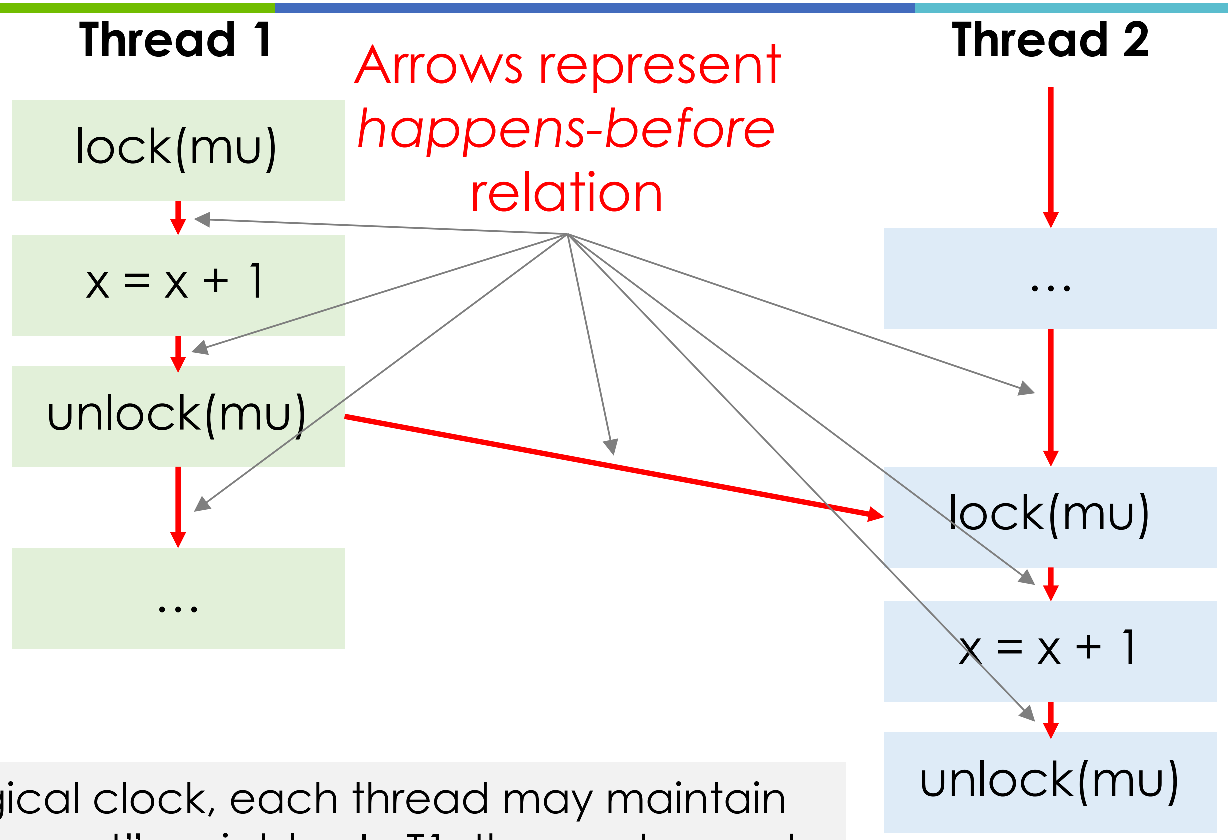
ev7 → ev8 → ev9

ev3 → ev7

ev2 → ev8

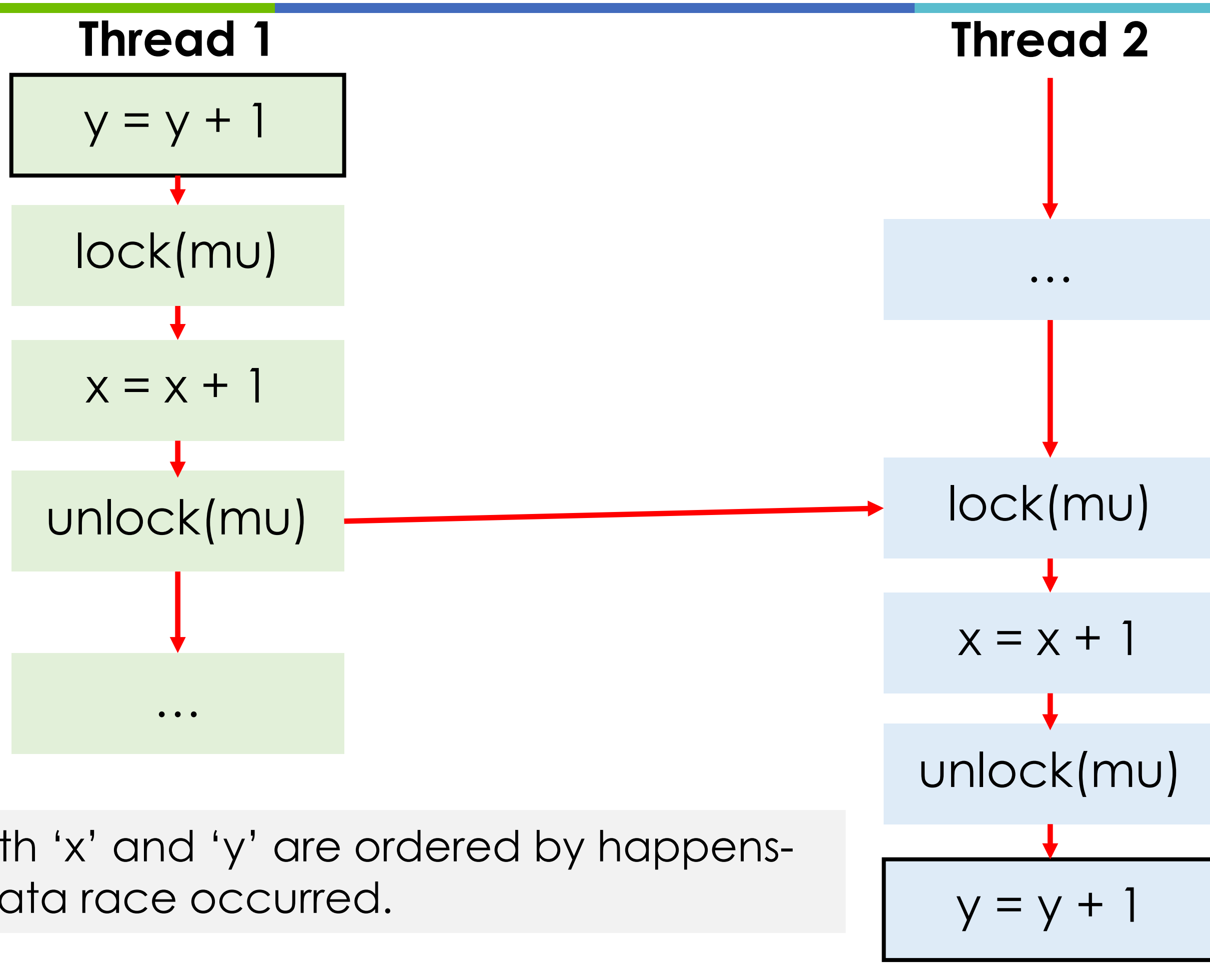
No Data Race

Example 1



Instead of a logical clock, each thread may maintain a "most recent event" variable. In T1, the most recent event is unlock(mu); when T2 executes lock(mu) the system can establish the happens-before relation.

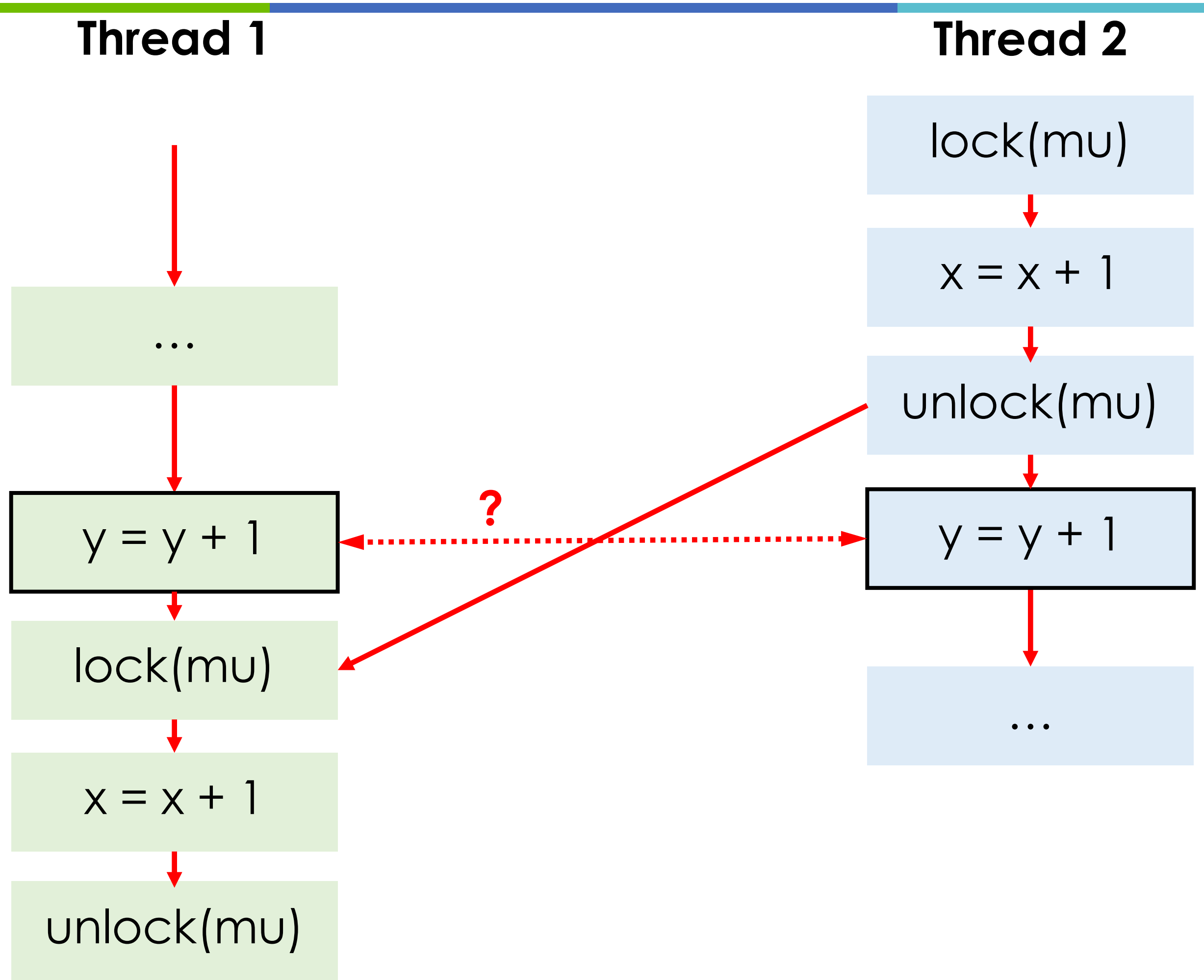
Example 2



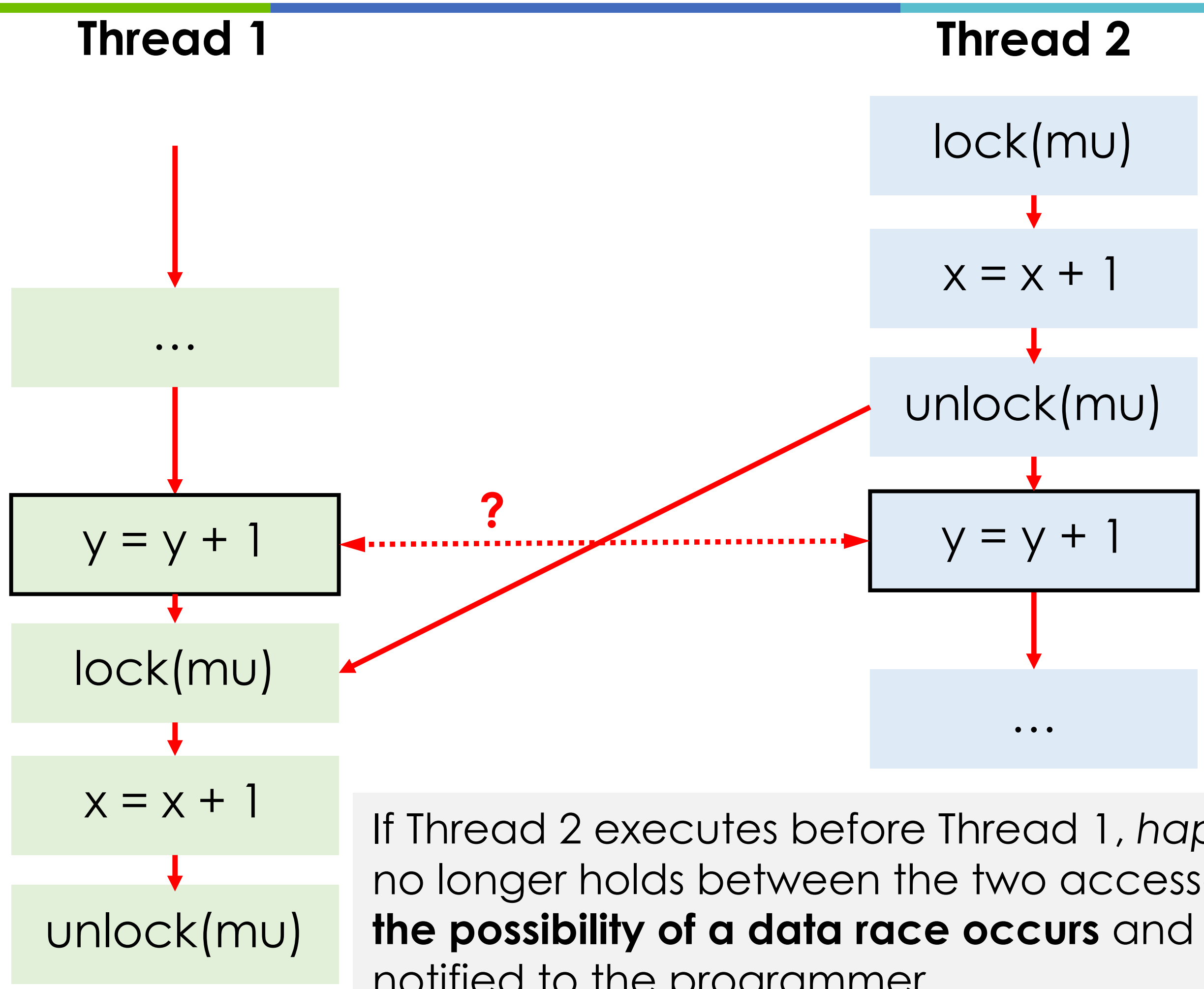
Accesses to both ‘x’ and ‘y’ are ordered by happens-before, so no data race occurred.

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

Example 3


Thread 1
Thread 2
lock(mu)
x = x + 1
unlock(mu)
...
y = y + 1
?
y = y + 1
lock(mu)
x = x + 1
unlock(mu)
...

Example 3



If Thread 2 executes before Thread 1, *happens-before* no longer holds between the two accesses to 'y', so **the possibility of a data race occurs** and should be notified to the programmer.

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

Accesses to 'bal' are concistently protected by 'balLock'.

Thread 1

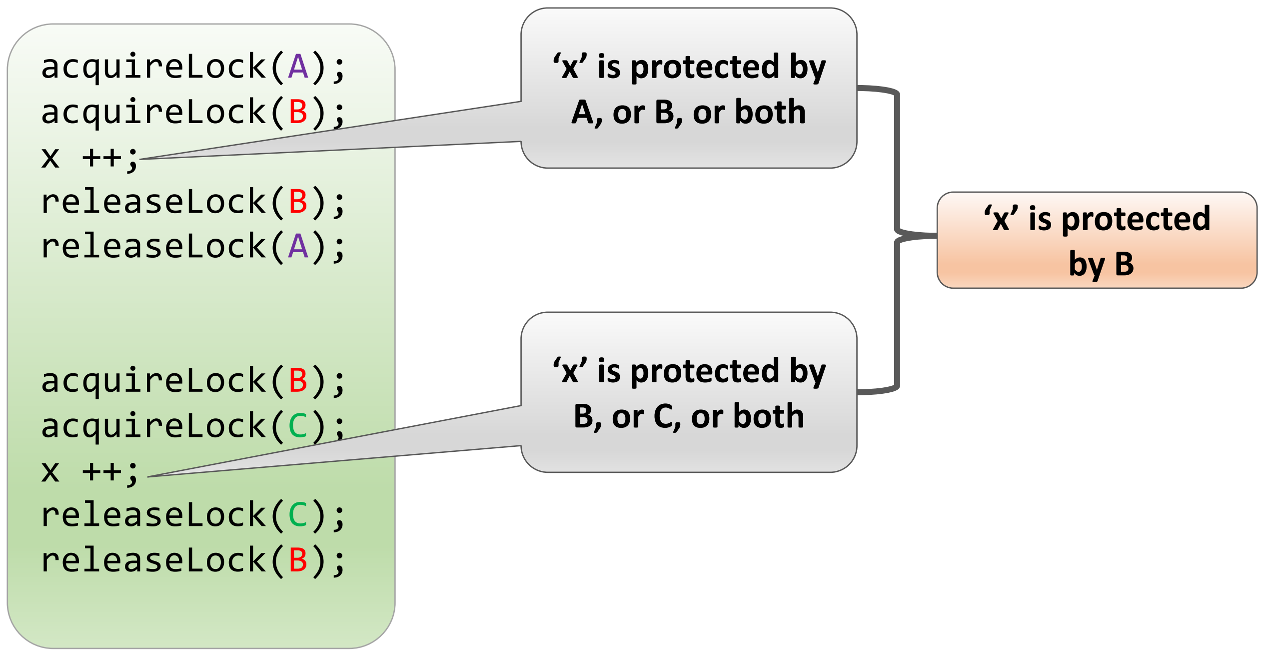
```
void Bank::Deposit(int a) {
    acquireLock(balLock);
    int t = bal;
    bal = t + a;
    releaseLock(balLock);
}
```

Thread 2

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

Approach

- 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:
  - `LocksHeld(t)` = set of locks held currently by thread t
    - Initially set to Empty
  - `LockSet(x)` = set of locks that could potentially be protecting x
    - Initially set to the universal set
- When thread 't' acquires lock 'l'
  - `LocksHeld(t) = LocksHeld(t) ∪ {l}`
- When thread 't' releases lock 'l'
  - `LocksHeld(t) = LocksHeld(t) \ {l}`
- When thread 't' accesses location 'x'
  - `LockSet(x) = LockSet(x) ∩ LocksHeld(t)`
- "Data race" warning if `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}`
`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}`
`lock (m1)` → ∪ →	`{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}
lock (m1)	{m1}	
lock(m2)	{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}`	
`lock(m2)`	`{m1, m2}`	
`v = v + 1`		`{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}	
lock(m2)	{m1, m2}	
v = v + 1		{m1, m2}
unlock(m2)	{m1}	
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

Another Example

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

Another Example

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

Another Example

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

Another Example

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

Another Example

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

Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	
lock(m2)	{m1, m2}	
v = v + 1		{m1, m2}
unlock(m2)	{m1}	
v = v + 2		{m1}
unlock(m1)	{ }	
lock(m2)	{m2}	
v = v + 1		{ } – **ALARM**
unlock(m2) → \ →	{ }	

Algorithm Guarantees

- No warnings => 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 => 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
  - www.microsoft.com/en-us/research/people/madanm/
  - williamstallings.com/OperatingSystems/
  - codex.cs.yale.edu/avi/os-book/OS9/slide-dir/

The END