

# **Program Testing and Analysis: Testing Concurrent Programs**

**Dr. Michael Pradel**

**Software Lab, TU Darmstadt**

# Outline

---

1. Introduction
2. Dynamic Data Race Detection
3. Testing Thread-Safe Classes
4. Exploring Interleavings

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
- *Fully Automatic and Precise Detection of Thread Safety Violations*, Pradel and Gross, PLDI 2012
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008

# Why Bother with Concurrency?

---

- The free lunch provided by **Moore's law** is over
  - CPU clock speeds stopped to increase around 2005
  - Instead, **multi-core processors** became mainstream
  - Need concurrent programs to make full use of the hardware
- Many real-world problems are **inherently concurrent**, e.g.,
  - Servers must handle multiple concurrent requests
  - Computations done on huge data often are **"embarrassingly parallel"**

# Why Bother with Concurrency?

■ The f

□ CF

□ Ins

□ Ne

ha

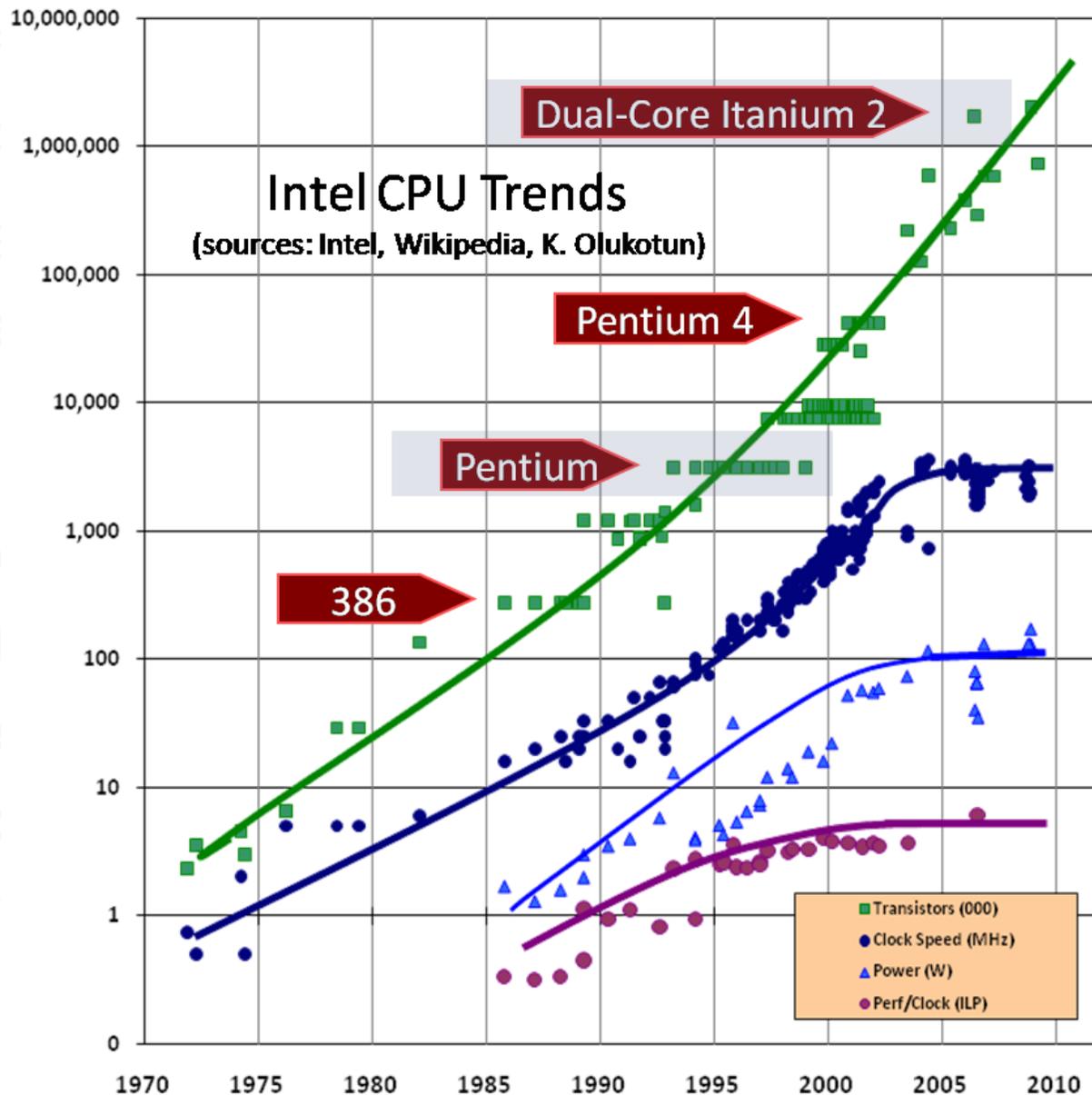
■ Many

concl

□ Se

□ Cc

"er



over  
and 2005  
stream  
of the

ests

# Why Bother with Concurrency?

---

- The free lunch provided by **Moore's law** is over
  - CPU clock speeds stopped to increase around 2005
  - Instead, **multi-core processors** became mainstream
  - Need concurrent programs to make full use of the hardware
- Many real-world problems are **inherently concurrent**, e.g.,
  - Servers must handle multiple concurrent requests
  - Computations done on huge data often are **"embarrassingly parallel"**

# Concurrency Styles

---

## ■ Message-passing

- Popular for large-scale scientific computing, e.g., MPI (message-passing interface)
- Used in **actor concurrency model**, e.g., popular in Erlang and Scala
- No shared memory (ideally), all communication via messages

## ■ Thread-based, shared memory

- Multiple concurrently executing threads
- All threads access the same shared memory
- Synchronize via **locks** and **barriers**

# Concurrency Styles

---

## ■ Message-passing

- Popular for large-scale scientific computing, e.g., MPI (message-passing interface)
- Used in **actor concurrency model**, e.g., popular in Erlang and Scala
- No shared memory (ideally), all communication via messages

## **Focus of this lecture**

## ■ Thread-based, shared memory

- Multiple concurrently executing threads
- All threads access the same shared memory
- Synchronize via **locks** and **barriers**

# Example

---

```
int a = 0, b = 0;
```

```
boolean r = false, t = false;
```



```
a = 1;
```

```
r = true;
```

```
t = r;
```

```
b = a;
```

**What does this program mean?**

**→ Behavior depends on thread interleaving**

$a = b = 0 ; r = t = \text{false}$

$a = 1$   
 $r = \text{true}$

$t = r$   
 $b = a$

①  
 $a = 1$   
 $r = \text{true}$   
 $t = r$   
 $b = a$

---

$t = \text{true}$   
 $b = 1$

②  
 $a = 1$   
 $t = r$   
 $b = a$   
 $r = \text{true}$

---

$t = \text{false}$   
 $b = 1$

③  
 $a = 1$   
 $t = r$   
 $r = \text{true}$   
 $b = a$

---

$t = \text{false}$   
 $b = 1$

④  
 $t = r$   
 $b = a$   
 $a = 1$   
 $r = \text{true}$

---

$t = \text{false}$   
 $b = 0$

⑤  
 $t = r$   
 $a = 1$   
 $r = \text{true}$   
 $b = a$

---

$t = \text{false}$   
 $b = 1$

⑥  
 $t = r$   
 $a = 1$   
 $b = a$   
 $r = \text{true}$

---

$t = \text{false}$   
 $b = 1$

$t = \text{true}$  implies  $b = 1$

# Sequential Consistency

---

**Assumption** made here:

Programs execute under **sequential consistency**

- **Program order** is preserved: Each thread's instructions execute in the specified order
- Shared memory behaves like a global array:  
**Reads and writes** are done **immediately**
- We assume sequential consistency for the rest of the lecture
- Many real-world platforms provide more complex semantics ("**memory models**")

# What Can Go Wrong?

---

## Common source of errors: **Data races**

- Two accesses to the same shared memory location
- At least one access is a write
- Ordering of accesses is non-deterministic

# Example

---

```
// bank account
```

```
int balance = 10;
```



```
// deposit money
```

```
int tmp1 = balance;
```

```
balance = tmp1 + 5;
```

```
// withdraw money
```

```
int tmp2 = balance;
```

```
balance = tmp2 - 7;
```

# Example

---

**Shared  
memory  
location**

```
// bank account
```

```
int balance = 10;
```

Thread 1

Thread 2

```
// deposit money
```

```
int tmp1 = balance;
```

```
balance = tmp1 + 5;
```

**Read**

```
// withdraw money
```

```
int tmp2 = balance;
```

```
balance = tmp2 - 7;
```

**Write**

# Example

---

```
// bank account
```

```
int balance = 10;
```



```
// deposit money
```

```
int tmp1 = balance;
```

```
balance = tmp1 + 5;
```

```
// withdraw money
```

```
int tmp2 = balance;
```

```
balance = tmp2 - 7;
```

**3 races**

# Example

---

```
// bank account
```

```
int balance = 10;
```



```
// deposit money
```

```
int tmp1 = balance;
```

```
balance = tmp1 + 5;
```

```
// withdraw money
```

```
int tmp2 = balance;
```

```
balance = tmp2 - 7;
```

**Quiz: What values can balance have after executing this code?**

# Example

---

```
// bank account
```

```
int balance = 10;
```



```
// deposit money
```

```
int tmp1 = balance;
```

```
balance = tmp1 + 5;
```

```
// withdraw money
```

```
int tmp2 = balance;
```

```
balance = tmp2 - 7;
```

**Possible outcomes:**

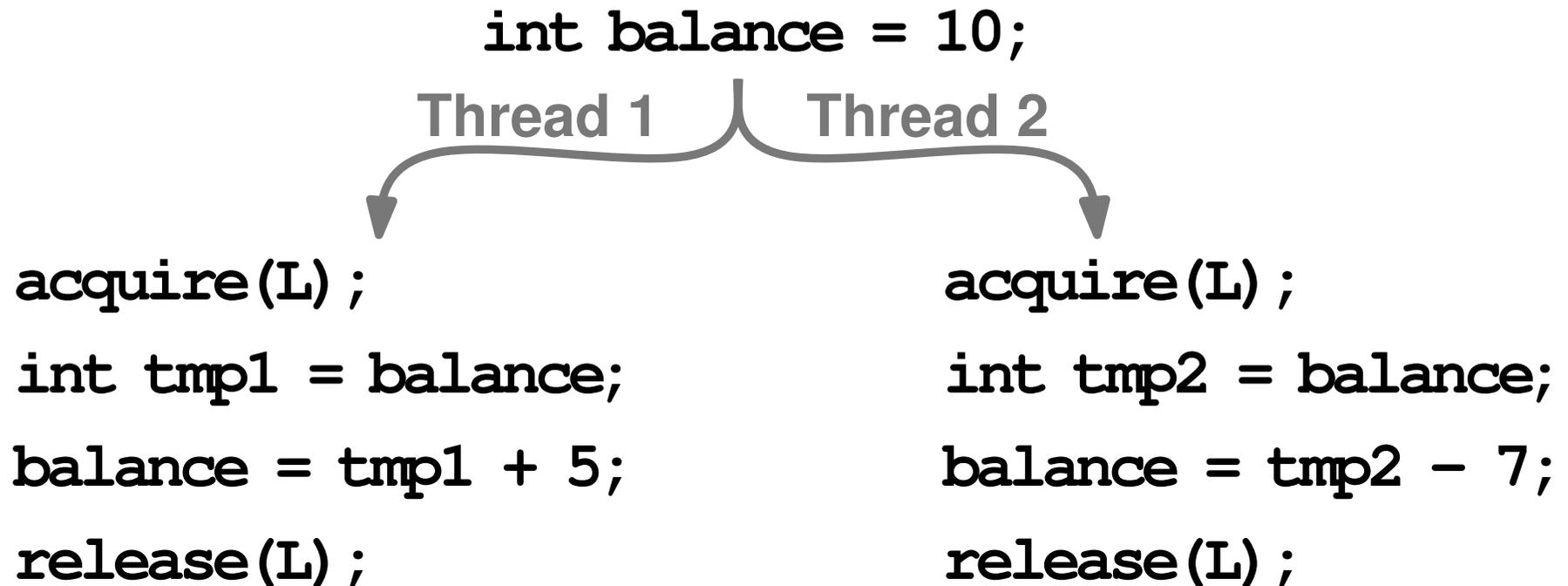
**balance may be 3, 8, and 15**

**But: Only 8 is correct**

# Avoiding Data Races

---

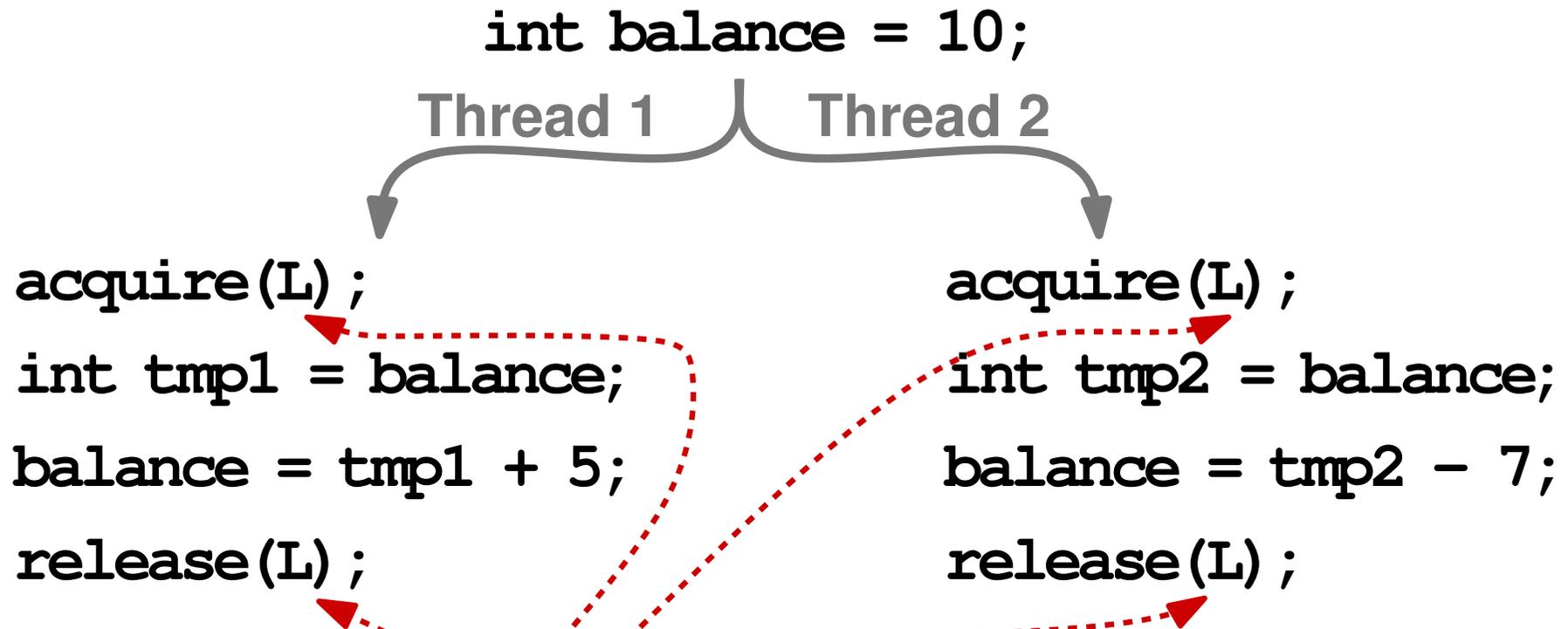
Use **locks** to ensure that **accesses** to shared memory **do not interfere**



# Avoiding Data Races

---

Use **locks** to ensure that **accesses** to shared memory **do not interfere**



**Same lock  $\Rightarrow$  Mutually  
exclusive critical sections**

# Avoiding Data Races

---

Use **locks** to ensure that **accesses** to shared memory **do not interfere**

```
        int balance = 10;
        Thread 1  Thread 2
        ↙         ↘
synchronized (L) {           synchronized (L) {
    int tmp1 = balance;       int tmp2 = balance;
    balance = tmp1 + 5;       balance = tmp2 - 7;
}                               }
```

(Java syntax)

# Outline

---

## 1. Introduction

## 2. Dynamic Data Race Detection



## 3. Testing Thread-Safe Classes

## 4. Exploring Interleavings

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
- *Fully Automatic and Precise Detection of Thread Safety Violations*, Pradel and Gross, PLDI 2012
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008

# Eraser: Data Race Detection

---

- Basic idea: Look for "unprotected" accesses to shared memory
- Assumption: All accesses to a shared memory location  $v$  should happen while holding the same lock  $L$ 
  - Consistent locking discipline
- Dynamic analysis that monitors all lock acquisitions, lock releases, and accesses to shared memory locations

# Lockset Algorithm (Simple Form)

---

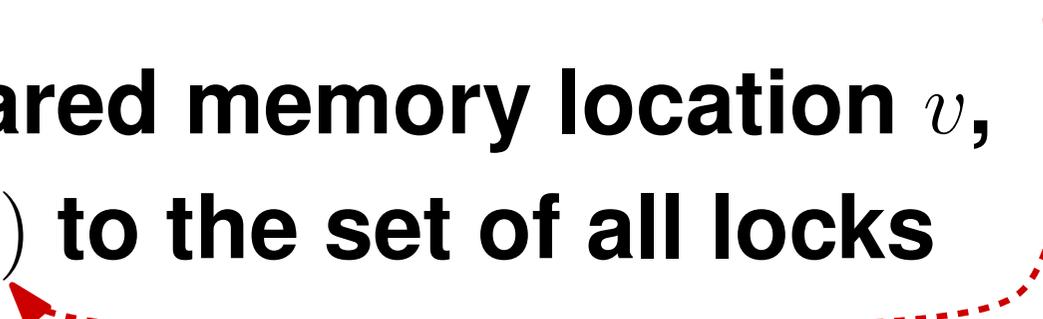
- **Let  $locksHeld(t)$  be the set of locks held by thread  $t$**
- **For each shared memory location  $v$ , initialize  $C(v)$  to the set of all locks**
- **On each access to  $v$  by thread  $t$** 
  - Set  $C(v) := C(v) \cap locksHeld(t)$
  - If  $C(v) = \emptyset$ , issue a warning

# Lockset Algorithm (Simple Form)

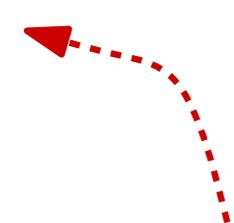
---

- Let  $locksHeld(t)$  be the set of locks held by thread  $t$
- For each shared memory location  $v$ , initialize  $C(v)$  to the set of all locks
- On each access to  $v$  by thread  $t$ 
  - Set  $C(v) := C(v) \cap locksHeld(t)$
  - If  $C(v) = \emptyset$ , issue a warning

Lockset



Lockset refinement



balance = 10

acquire (L1)  
 tmp1 = balance  
 balance = tmp1 + a  
 release (L1)

tmp2 = balance  
 acquire (L2)  
 balance = tmp2 - b  
 release (L2)

Instruction	locksHeld	C(balance)
balance = 10	{}	{L1, L2}
acquire (L1)	{L1}	{L1}
tmp1 = balance		{L1}
balance = tmp1 + a		{L1}
release (L1)	{}	{}
tmp2 = balance		
acquire (L2)	{L2}	
balance = tmp2 - b		
release (L2)	{}	

→ warning

# Simple Lockset is Too Strict

---

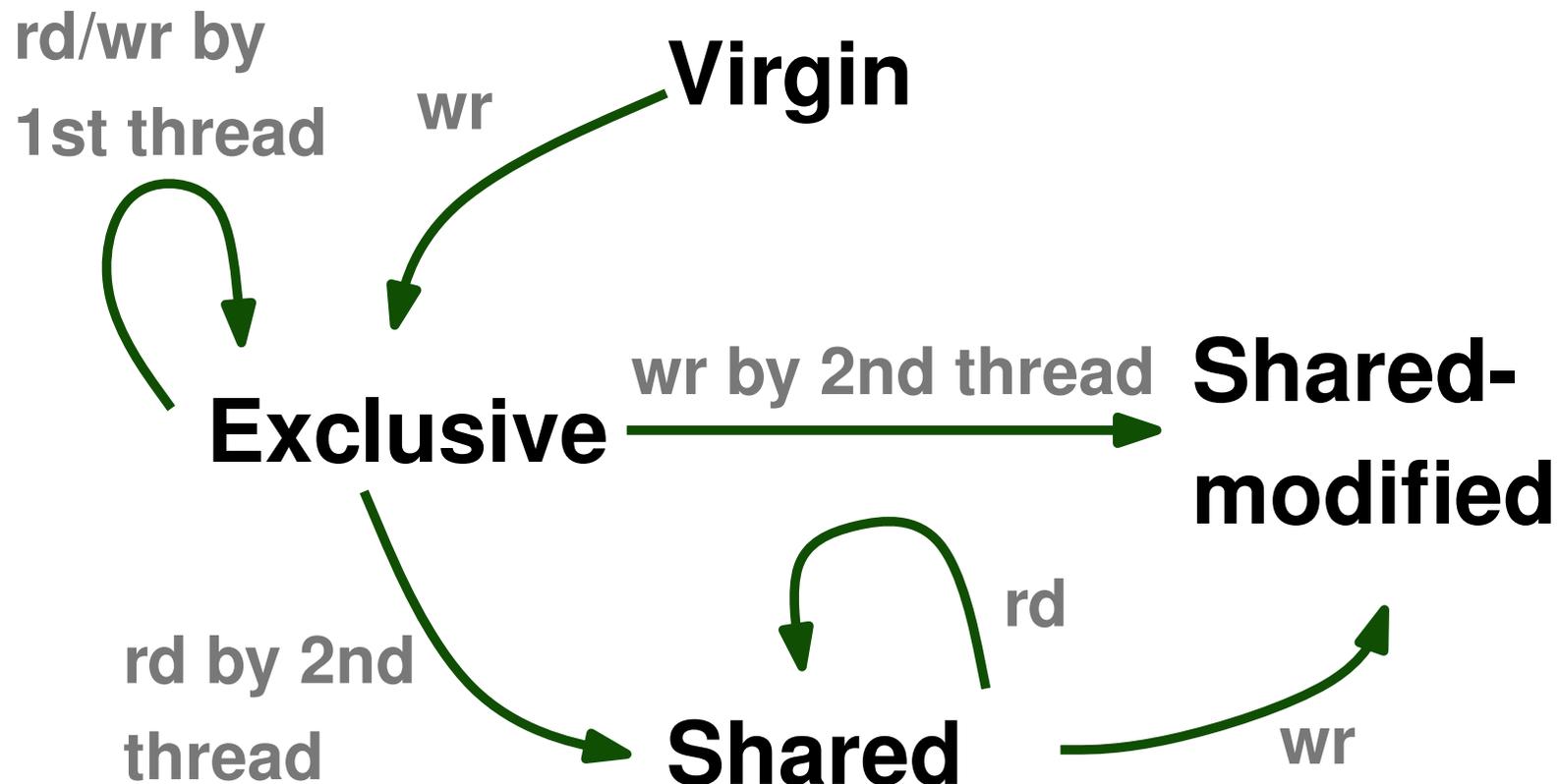
Simple lockset algorithm produces **false positives** for

- variables initialized without locks held
- read-shared data read without locks held
- read-write locking mechanisms  
(producer-consumer style)

# Refining the Lockset Algorithm

---

- Keep **state** of each **shared memory location**
- Issue warnings only in the Shared-modified state



# Summary: Eraser

---

- **Dynamic analysis** to detect data races
- **Assumes consistent locking discipline**
- **Limitations**
  - **May report false positives** when locks are acquired inconsistently but correctly
  - **May miss data races** because it does not consider all possible interleavings

# Outline

---

## 1. Introduction

## 2. Dynamic Data Race Detection

## 3. Testing Thread-Safe Classes

## 4. Exploring Interleavings

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
- *Fully Automatic and Precise Detection of Thread Safety Violations*, Pradel and Gross, PLDI 2012
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008