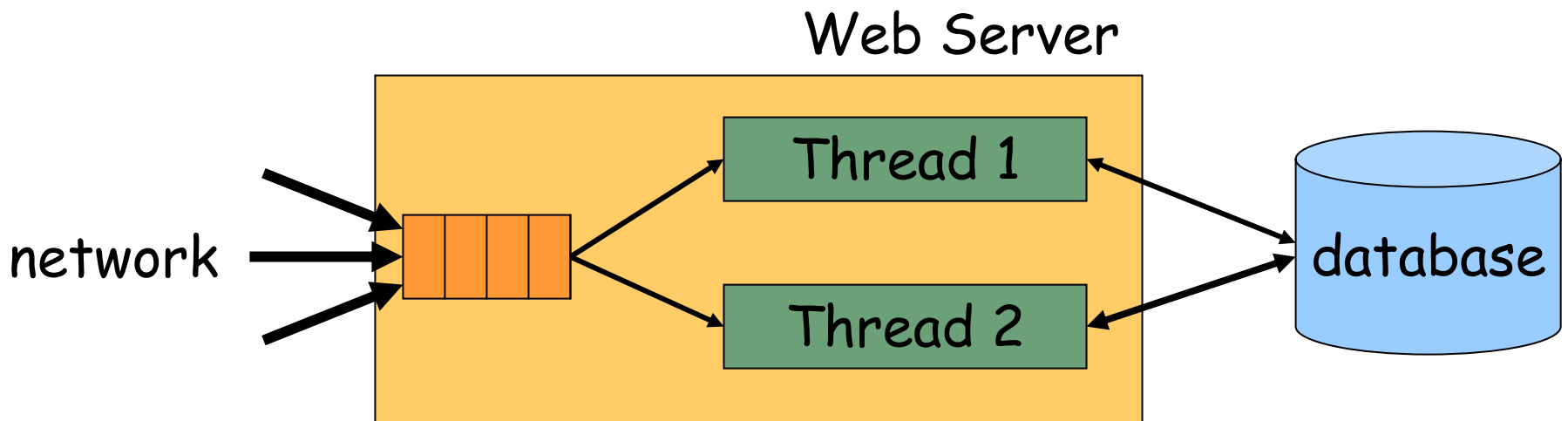


Type-Based Race Detection for Java

Programming With Threads

- Decompose program into pieces that can run in parallel
- Advantages
 - exploit multiple processors
 - threads make progress, even if others are blocked



Multithreaded Program Execution

Thread 1

Thread 2

...

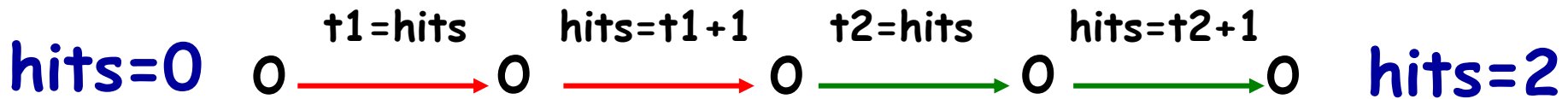
```
t1 = hits;  
hits = t1 + 1;
```

...

...

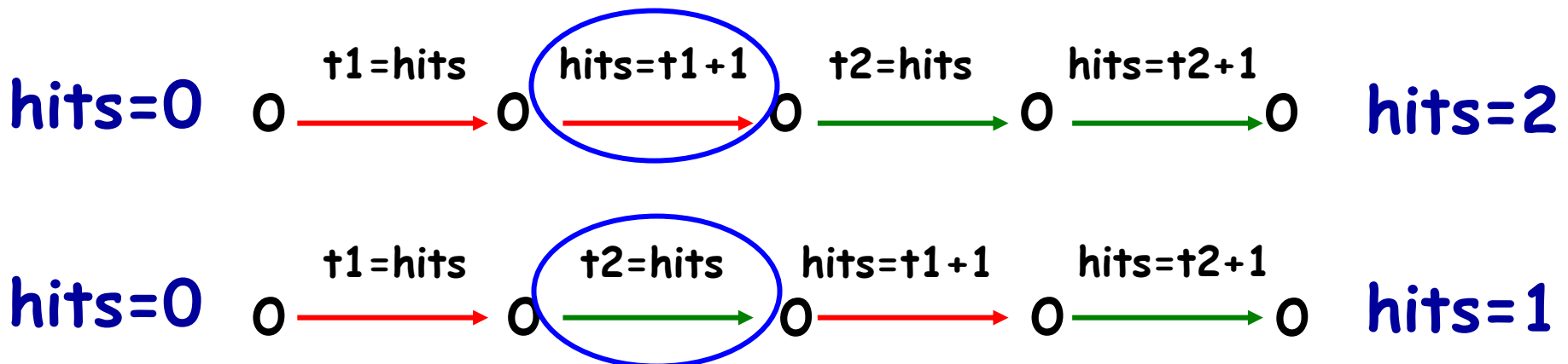
```
t2 = hits;  
hits = t2 + 1;
```

...



Multithreaded Program Execution

A *race condition* occurs if two threads access a shared variable at the same time, and at least one of the accesses is a write



Preventing Race Conditions Using Locks

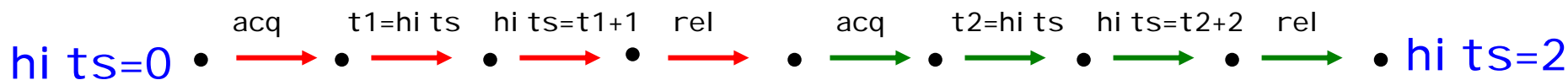
- Lock can be held by at most one thread at a time
- Race condition prevention
 - associate a lock with each shared variable
 - acquire lock before accessing variable

Thread 1

```
synchronized(lock) {  
    int t1 = hi ts;  
    hi ts = t1 + 1  
}
```

Thread 2

```
synchronized(lock) {  
    int t2 = hi ts;  
    hi ts = t2 + 2  
}
```



Problem With Current Practice

- Locking discipline is not enforced
 - inadvertent programming errors cause races
- Race conditions are insidious bugs
 - non-deterministic, timing dependent
 - data corruption, crashes
 - difficult to detect, reproduce, eliminate
- Linux 2.4 log has 36 synchronization bug fixes

Reliable Multithreaded Software

- Correctness Problem
 - does program behaves correctly for *all inputs* and *all interleavings*?
 - very hard to ensure with testing
- Use static checkers
 - type systems target sequential programs
 - **need type systems for multithreaded programs!**

Use Type System to Ensure Race Freedom

- Static type system prevents race conditions
- Programmer specifies synchronization discipline
 - lock protecting each field
 - locks held on entry to each method
- Type checker checks synchronization discipline
 - checks field accessed only when lock held
 - checks for *all inputs* and *all interleavings*

Synchronized Bank Account

```
class Account {  
    private int balance = 0;  
    private void update(int x) {  
        balance = x;  
    }  
    public void deposit(int n) {  
        synchronized(this) {  
            update(balance + n);  
        }  
    }  
}
```

Thread 1

acct. deposit(100);

Thread 2

acct. deposit(100);

Annotated Account

```
class Account {
    private int balance = 0    /*# guarded_by this */;
    private void update(int x) /*# requires this */ {
        balance = x;
    }
    public void deposit(int n) {
        synchronized(this) {
            update(balance + n);
        }
    }
}
```

Annotated Account

```
class Account {
  private int balance = 0
  private void update(int x)
    balance = x;
}
public void deposit(int n) {
  synchronized(this) {
    update(balance + n);
  }
}
```

```
guarded_by this ;
requires this {
```

Annotated Account

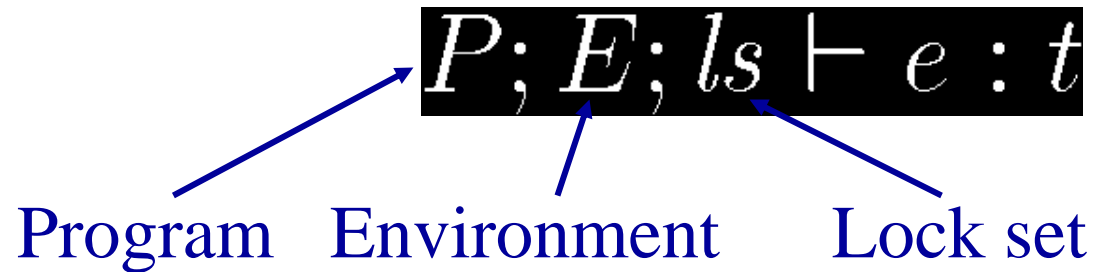
```
class Account {  
    private int balance = 0  
    private void update(int x)    guarded_by this ;  
        balance = x;            requires this {  
    }  
    public void deposit(int n) {  
        synchronized(this) {  
            update(balance + n);  
        }  
    }  
}
```

- Lock names are constant expressions:
 - `this`
 - `x.f` (if `x` is a constant expression and `f` a final field)
 - `java.lang.System.out` (static final field)

Tracking Lock Sets

```
class Account {  
    private int balance = 0; guarded_by this;  
    private void update(int x) requires this {  
        lockset is {this}  
        balance = x; this ∈ lockset ? Yes  
    }  
  
    public void deposit(int n) {  
        lockset is empty  
        synchronized(this) {  
            lockset is {this}  
            update(balance + n); {this} ⊆ lockset ? Yes  
        }  
    }  
}
```

Typing Judgment



Typing Rules

- Thread creation

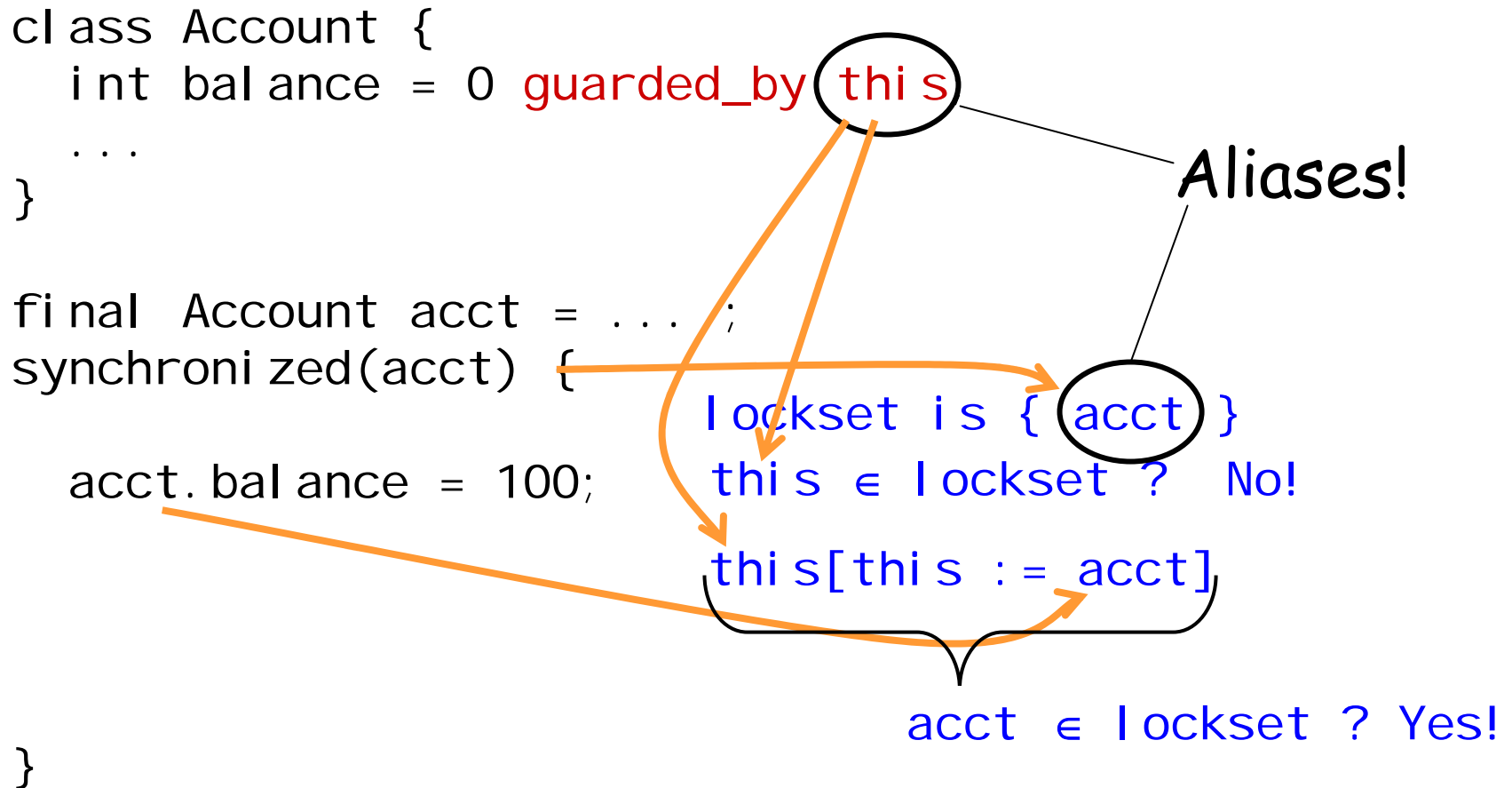
$$\frac{P; E; \emptyset \vdash e : t}{P; E; ls \vdash \text{fork } e : \text{int}}$$

- Lock acquisition

$$\frac{\begin{array}{l} P; E \vdash_{\text{final}} e_1 : c \\ P; E; ls \cup \{e_1\} \vdash e_2 : t \end{array}}{P; E; ls \vdash \text{synchronized } e_1 \text{ in } e_2 : t}$$

lock is constant
add to lock set

Handling Aliases Using Substitutions



Externally Synchronized Account

```
class Account<Object Lock> {  
    private int balance = 0 guarded_by Lock;  
    private void update(int x) requires Lock {  
        balance = x;  
    }  
    public void deposit(int n) requires Lock {  
        update(balance + n);  
    }  
}  
  
final Object aLock = new Object();  
Account<aLock> acct = new Account<aLock>();  
synchronized(aLock) { acct.deposit(100); } // ok  
  
acct.deposit(100); // error
```

Soundness of the Type System

- Soundness Guarantee:
 - well-typed programs do not have race conditions
- Some good programs have "benign races"
 - allow program to escape type system

```
class Account {  
    private int balance guarded_by this;  
    public Account(int n) {  
        balance = n; //# no_warn  
    }  
}
```

Race-Free Type System Features

- Guarded fields
- Lock sets
- Aliases
- Parameterized classes
- Escapes
- Dependant types
- Subtyping
- Thread local analysis
- Constant analysis
- Arrays, ...

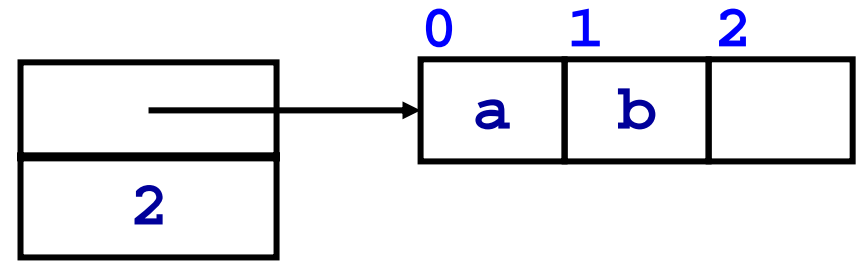
Validation of Race Condition Checker

Program	Size (lines)	Number of annotations	Annotation time (hrs)	Races Found
Hashtable	434	60	0.5	0
Vector	440	10	0.5	1
java.io	16,000	139	16.0	4
Ambit	4,500	38	4.0	4
WebL	20,000	358	12.0	5

Validation of Race Condition Checker

Program	Size (lines)	Number of annotations	Annotation time (hrs)	Races Found
Hashtable	434	60	0.5	0
Vector	440	10	0.5	1
java.io	16,000	139	16.0	4
Ambit	4,500	38	4.0	4
WebL	20,000	358	12.0	5
TLC	53,500	n/a	n/a	4
orange	28,000	n/a	n/a	1
red	450,000	n/a	n/a	~20

java.util.Vector



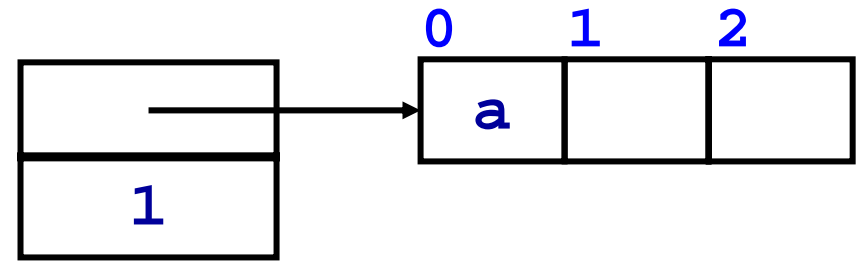
```
class Vector {
    Object elementData[] guarded_by this;
    int elementCount      guarded_by this;

    int lastIndexOf(Object elem) { RACE
        return lastIndexOf(elem, elementCount - 1);
    }

    synchronized int lastIndexOf(Object elem, int n) {
        for (int i = n ; i >= 0 ; i--)
            if (elem.equals(elementData[i])) return i;
        return -1;
    }

    synchronized boolean remove(int index) { ... }
    synchronized void trimToSize() { ... }
}
```

java.util.Vector



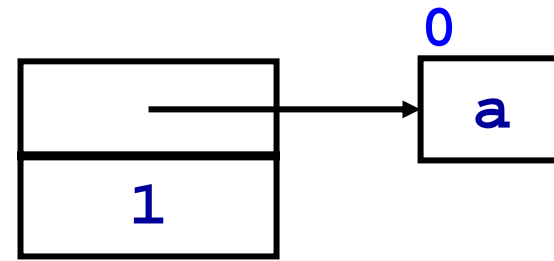
```
class Vector {
    Object elementData[] guarded_by this;
    int elementCount      guarded_by this;

    int lastIndexOf(Object elem) { RACE
        return lastIndexOf(elem, elementCount - 1);
    }

    synchronized int lastIndexOf(Object elem, int n) {
        for (int i = n ; i >= 0 ; i--)
            if (elem.equals(elementData[i])) return i;
        return -1;
    }

    synchronized boolean remove(int index) { ... }
    synchronized void trimToSize() { ... }
}
```

java.util.Vector



```
class Vector {
    Object elementData[] guarded_by this;
    int elementCount      guarded_by this;

    int lastIndexOf(Object elem) { RACE
        return lastIndexOf(elem, elementCount - 1);
    }

    synchronized int lastIndexOf(Object elem, int n) {
        for (int i = n ; i >= 0 ; i--)
            if (elem.equals(elementData[i])) return i;
        return -1;
    }
    }

    synchronized boolean remove(int index) { ... }
    synchronized void trimToSize() { ... }
}
```