

# Randomized Active Atomicity Violation Detection in Concurrent Programs

Chang-Seo Park  
Koushik Sen  
UC Berkeley EECS Dept.

FSE 16 11/12/2008

# Outline

---

- ▶ **Overview of Concepts & Motivation**
  - ▶ Correctness of Concurrent Programs
  - ▶ Random testing
  - ▶ Bug pattern for atomicity violations
- ▶ **Randomized Active Testing (for Atomicity)**
- ▶ **Evaluation**
- ▶ **Limitations**
- ▶ **Conclusion**

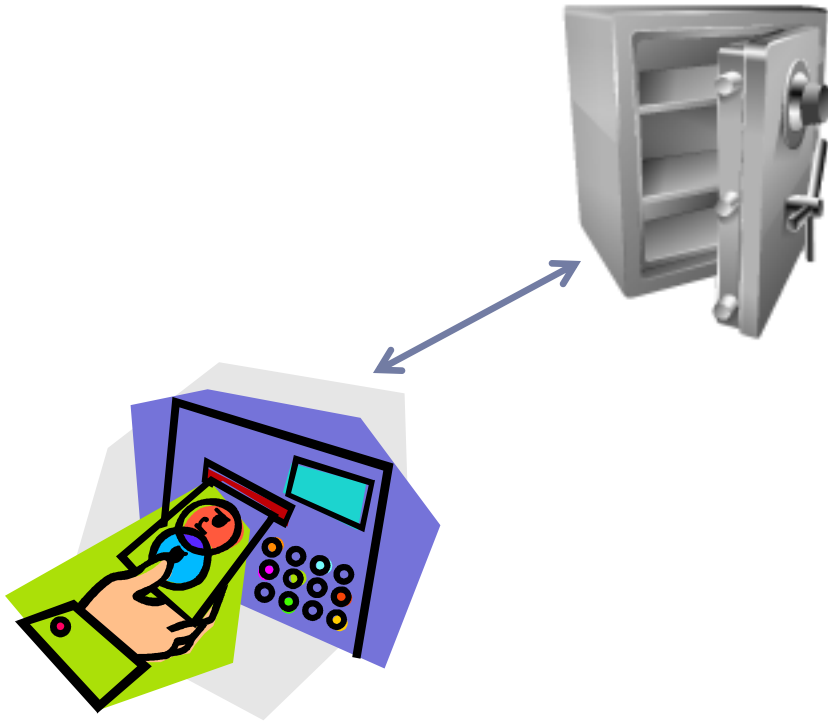
# Concurrent Programs

---

- ▶ **Multi-threaded Shared Memory**
- ▶ **Correct concurrent code is**
  - ▶ Correct on all legal inputs
  - ▶ Correct on all possible schedules
- ▶ **Need to explicitly reason about**
  - ▶ Interference among threads
  - ▶ Scheduling issues
- ▶ **Bugs due to concurrency**
  - ▶ Data races
  - ▶ Deadlocks
  - ▶ Atomicity violations

# Concurrency Bugs in Real Life

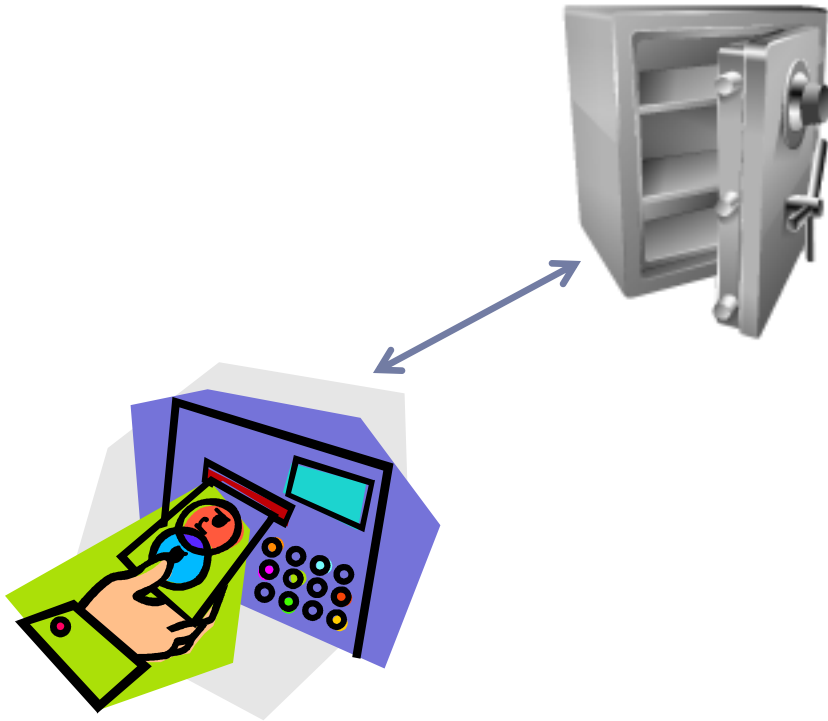
---



withdraw(50)

# Concurrency Bugs in Real Life

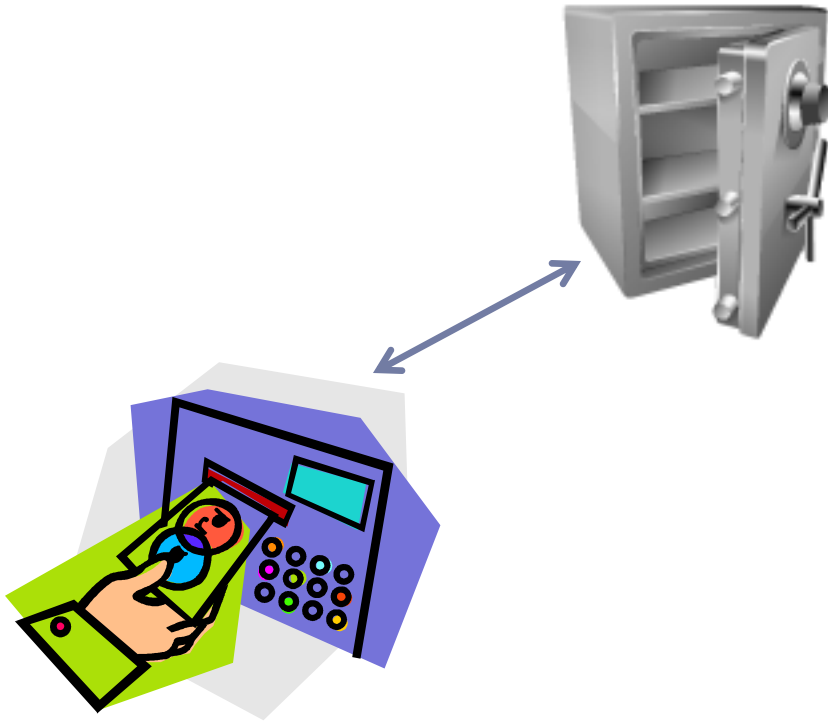
---



```
balance = getBalance()  
if(balance >= 50)  
    withdraw(50)
```

# Concurrency Bugs in Real Life

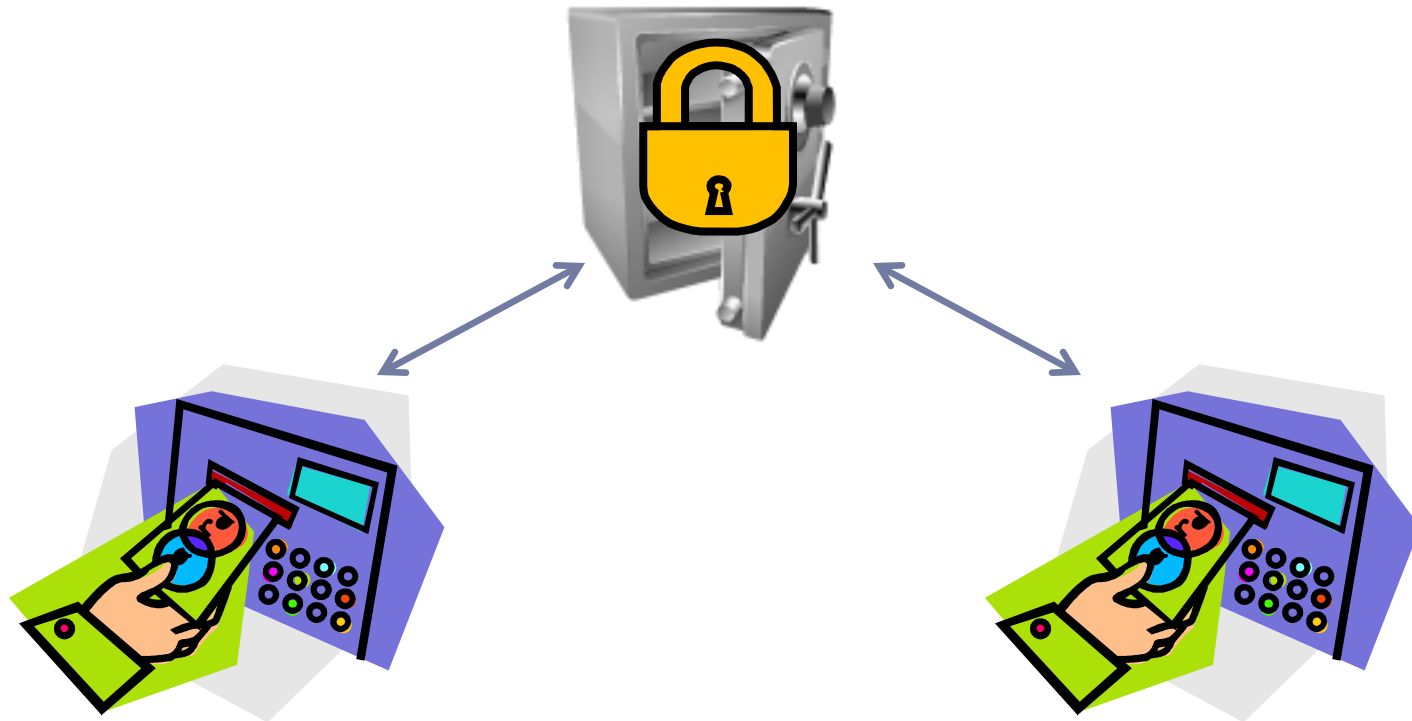
---



```
{pre: balance >= 0}  
balance = getBalance()  
if(balance >= 50)  
    withdraw(50)  
{post: balance >= 0}
```

# Concurrency Bugs in Real Life

---

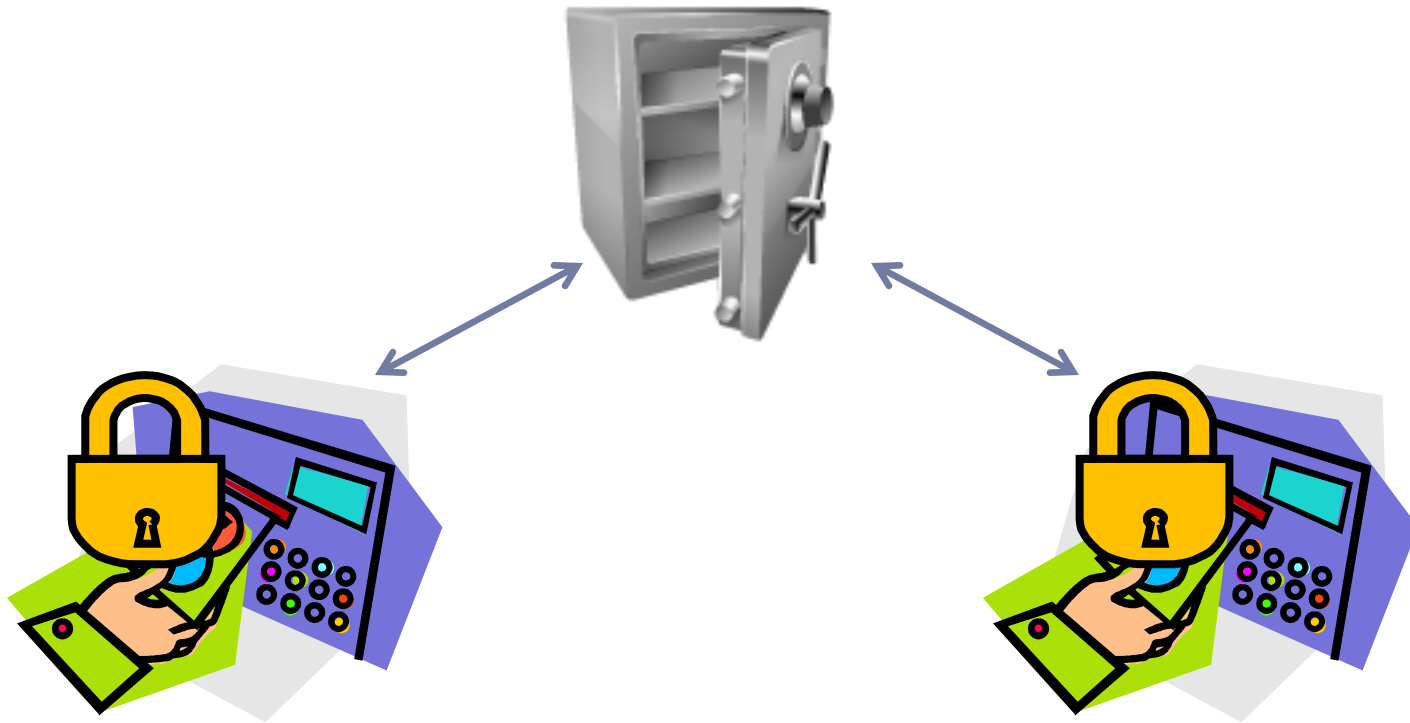


```
{pre: balance >= 0}  
balance = getBalance()  
if(balance >= 50)  
    withdraw(50)  
{post: balance >= 0}
```

```
{pre: balance >= 0}  
balance = getBalance()  
if(balance >= 50)  
    withdraw(50)  
{post: balance >= 0}
```

# Concurrency Bugs in Real Life

---

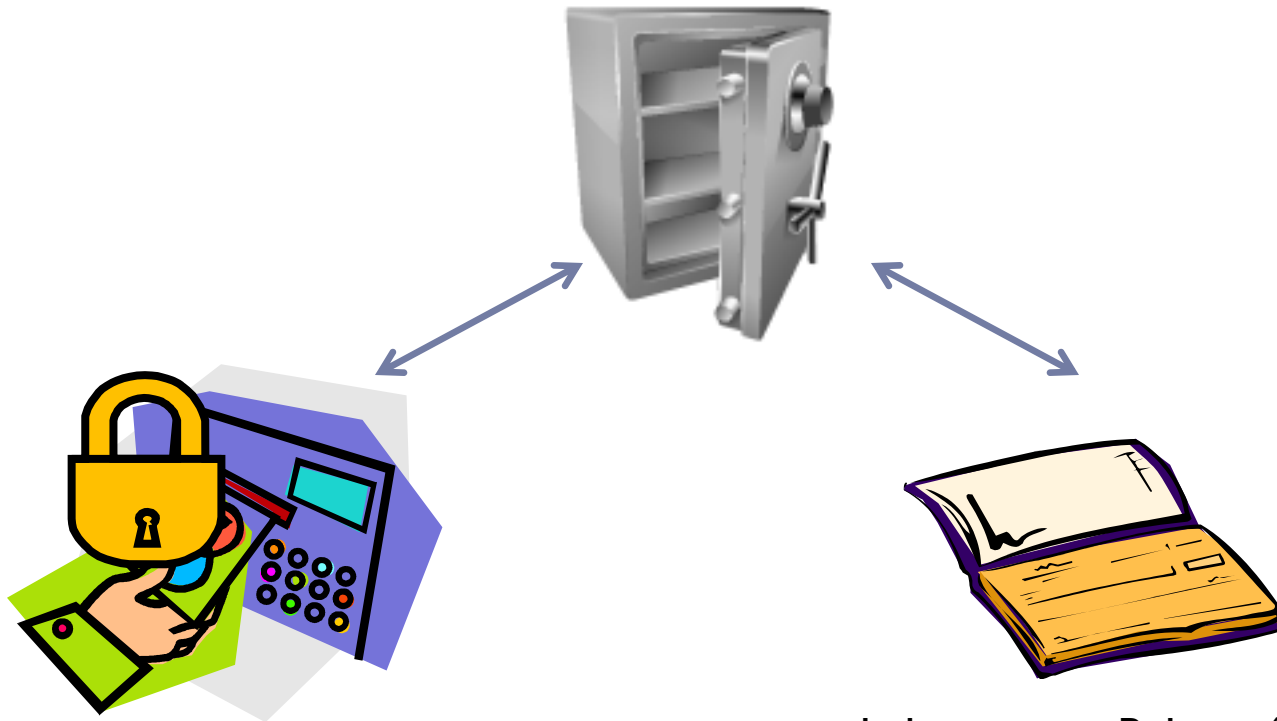


```
{pre: balance >= 0, hasAccLock}  
balance = getBalance()  
if(balance >= 50)  
    withdraw(50)  
{post: balance >= 0}
```

```
{pre: balance >= 0, hasAccLock}  
balance = getBalance()  
if(balance >= 50)  
    withdraw(50)  
{post: balance >= 0}
```

# Concurrency Bugs in Real Life

---

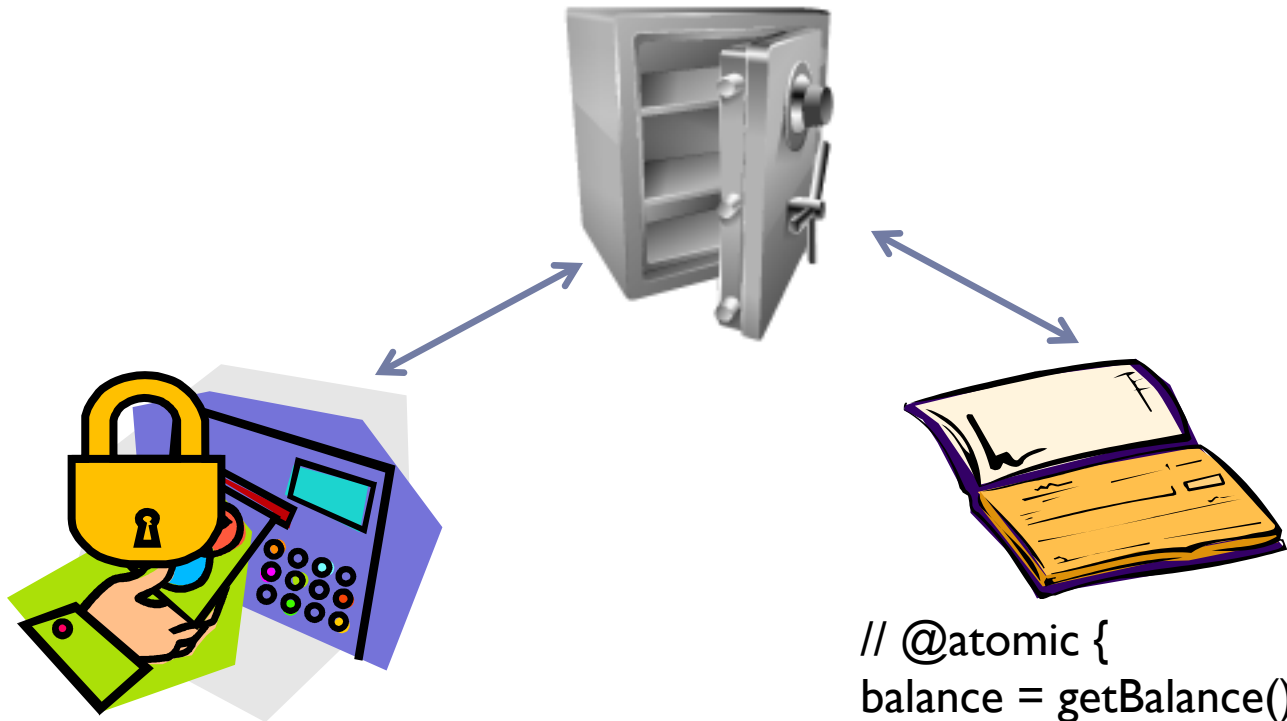


```
{pre: balance >= 0, hasAccLock}  
balance = getBalance()  
if(balance >= 50)  
    withdraw(50)  
{post: balance >= 0}
```

```
balance = getBalance()  
if(balance >= 100)  
    writeCheck(100)  
    [long delay]  
    cashOut(100)
```

# Concurrency Bugs in Real Life

---



```
// @atomic {  
balance = getBalance()  
if(balance >= 50)  
    withdraw(50)  
// @}
```

```
// @atomic {  
balance = getBalance()  
if(balance >= 100)  
    writeCheck(100)  
    [long delay]  
    cashOut(100)  
// @}
```

# Atomicity

---

- ▶ A block of code is **atomic** if
  - ▶ All operations inside execute “together”
  - ▶ No other thread can affect its operations
  - ▶ Other threads either see the shared memory state of either before the block is executed or after

# Other examples

---

## ▶ Producer / Consumer

```
// @atomic
if(!buffer.empty()) {
    ...
    buffer.consume();
}
```

## ▶ java.lang.StringBuffer

```
// @atomic
append(String str) {
    int len = str.length();
    ...
    str.getChars(0, len, value, count);
}
```

# Finding Concurrency Bugs

---

## ▶ Model Checking

- ▶ [Hatcliff et al;VMCAI'04], [Flanagan;SPIN'04]
- ▶ Try out all inputs and thread schedules
- ▶ Too expensive for large programs

## ▶ Type Systems

- ▶ [Lipton; CACM'75], [Flanagan et al; PLDI'03]
- ▶ Annotation burden for programmers

## ▶ Stress Testing

- ▶ Run the program with varying inputs multiple times
- ▶ Low coverage without control of scheduler

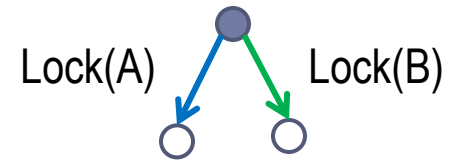
# Dynamic Random Schedule Testing

---

- ▶ **For every program state**
  - ▶ Pick a random enabled thread
  - ▶ Execute transition
  - ▶ Repeat until no live threads or error observed
- ▶ **Easy to implement**
- ▶ **Small number of threads to choose from at each state**

**B = 100**

```
1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
→ 6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```



### Thread T1

```
→ 11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

### Thread T2

```
21: A.withdraw(50);
```

**B = 100**

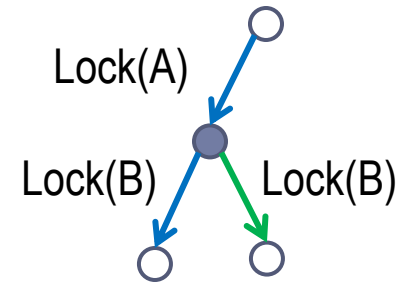
```
1: class Account {
2:   Integer B = 100;
→ 3:   synchronized(B) getBalance() {
4:     return B;
5:   }
→ 6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

### Thread T2

```
21: A.withdraw(50);
```



**B = 100**

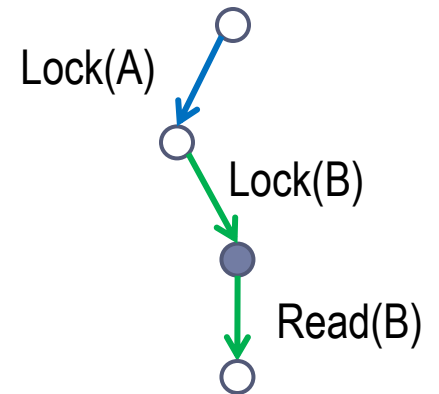
```
1: class Account {
2:   Integer B = 100;
→ 3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
→ 7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

### Thread T2

```
21: A.withdraw(50);
```



**B = 100**

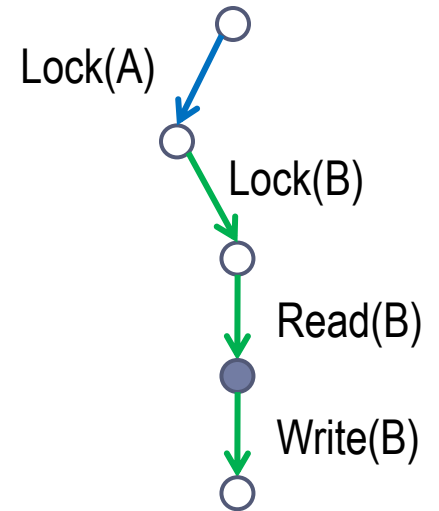
```
1: class Account {
2:   Integer B = 100;
→ 3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
→ 7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

### Thread T2

```
21: A.withdraw(50);
```



**B = 50**

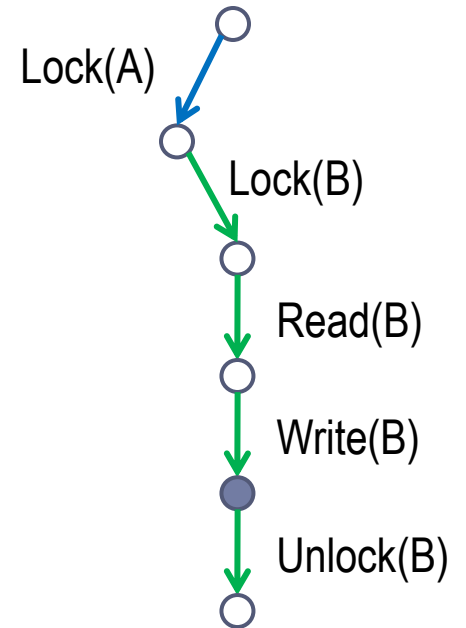
```
1: class Account {
2:   Integer B = 100;
→ 3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
→ 9:   }
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

### Thread T2

```
21: A.withdraw(50);
```



**B = 50**

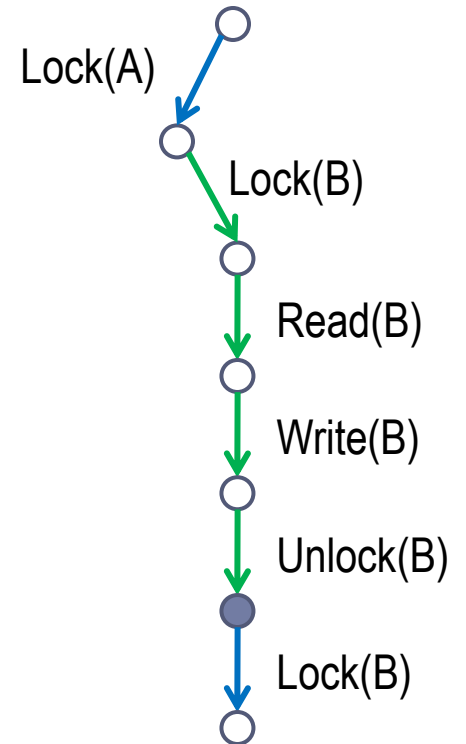
```
1: class Account {  
2:   Integer B = 100;  
→ 3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
6:   synchronized(B) withdraw(int amt) {  
7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

### Thread T2

```
21: A.withdraw(50);
```



**B = 50**

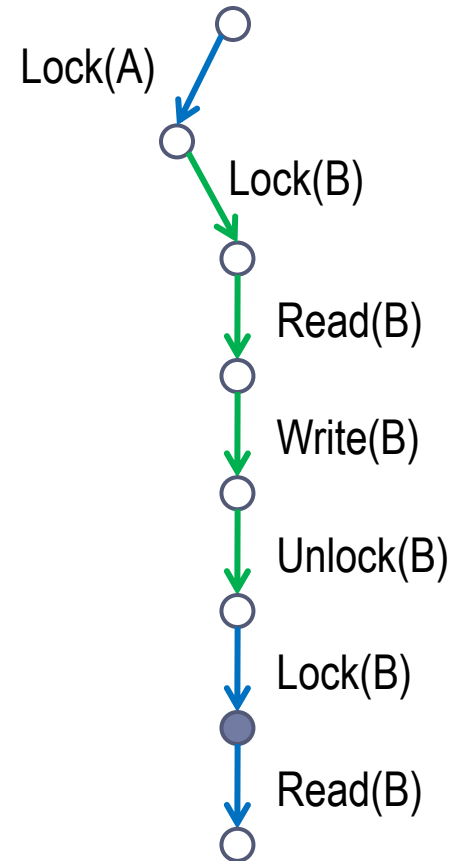
```
1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

### Thread T2

```
21: A.withdraw(50);
```



**B = 50**

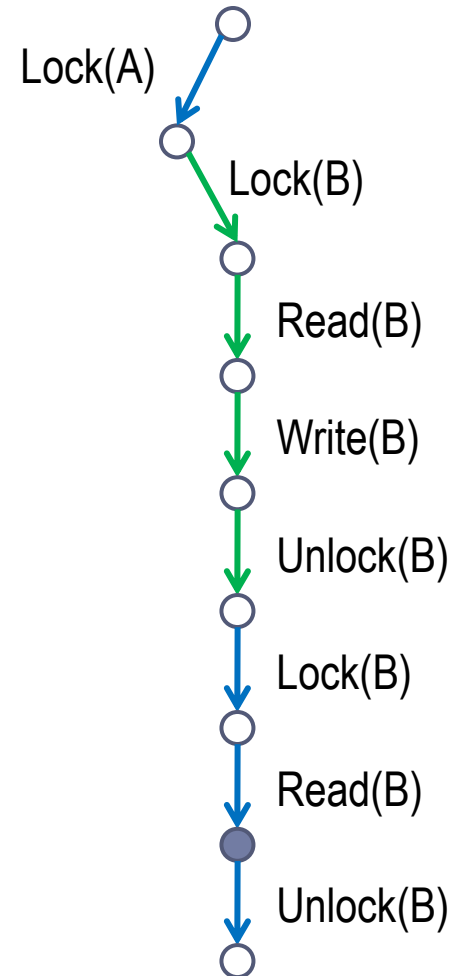
```
1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

### Thread T2

```
21: A.withdraw(50);
```



**B = 50**

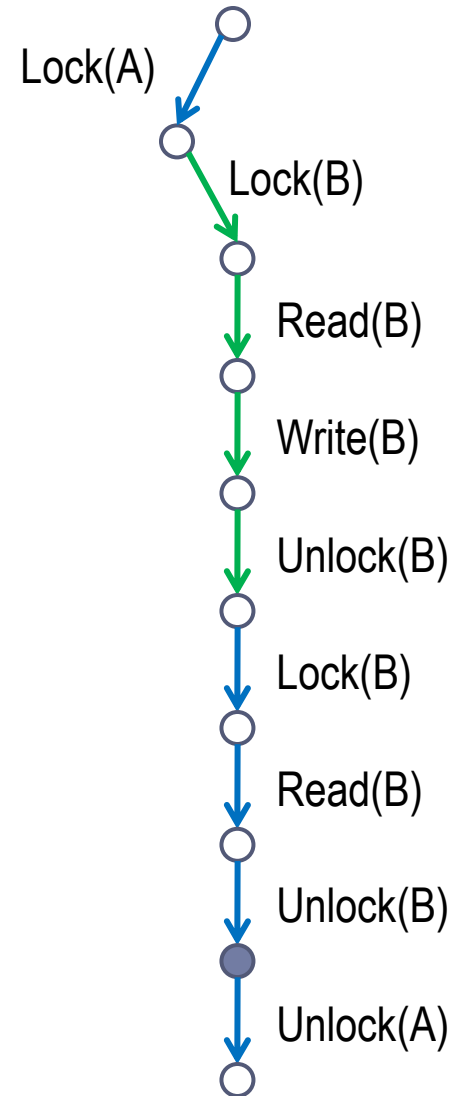
```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
6:   synchronized(B) withdraw(int amt) {  
7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
→14: }
```

### Thread T2

```
21: A.withdraw(50);
```



**B = 50**

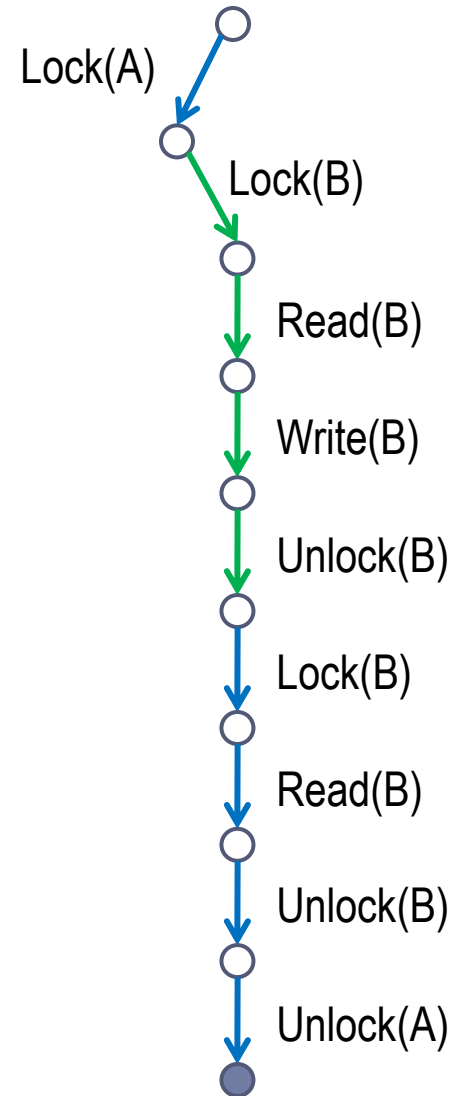
```
1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

### Thread T2

```
21: A.withdraw(50);
```



**B = 50**

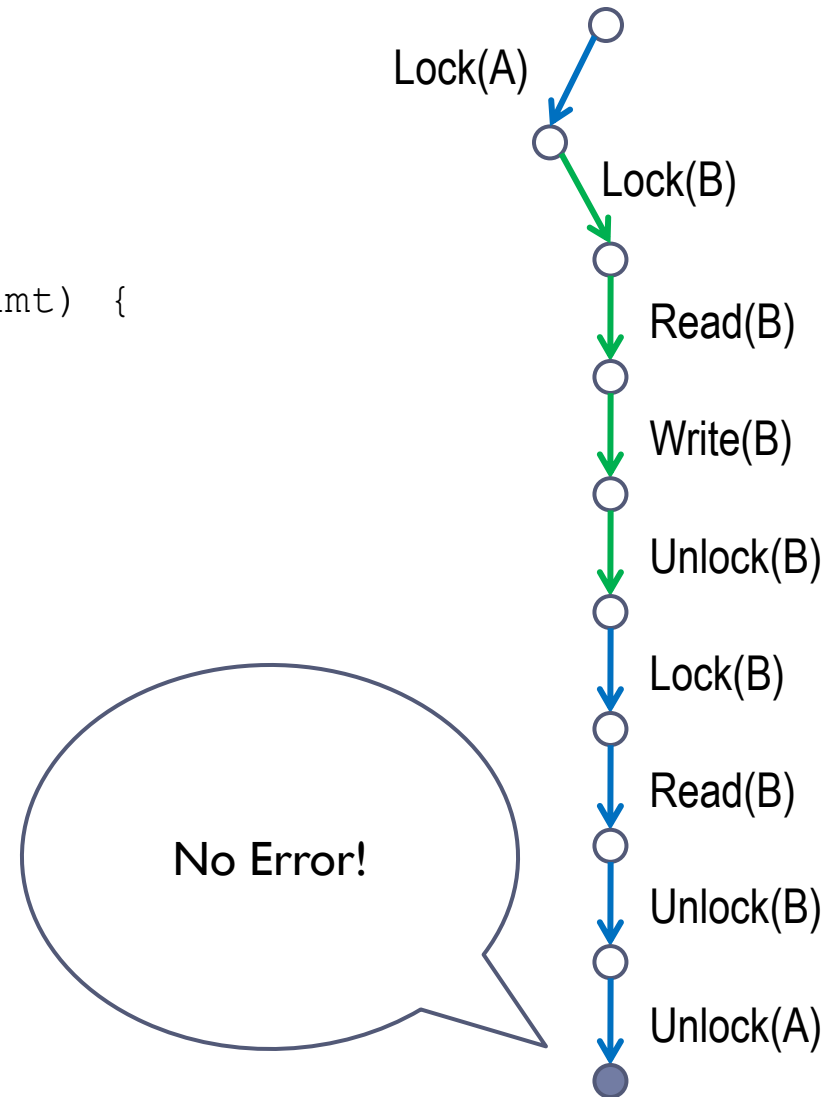
```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
6:   synchronized(B) withdraw(int amt) {  
7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

### Thread T1

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

### Thread T2

```
21: A.withdraw(50);
```

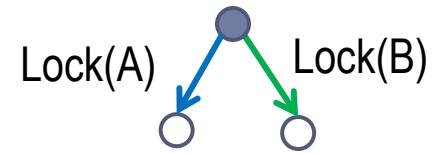


# Pros and Cons of Random Testing

---

- ▶ **Schedules are reproducible**
  - ▶ Use same seed for random number generator
  - ▶ Keep other non-deterministic inputs constant
- ▶ **Good at sampling various possible schedules**
  - ▶ May not be good enough to uncover bugs
  - ▶ Was the above trace bug free?

**B = 100**



```
1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
→ 6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```

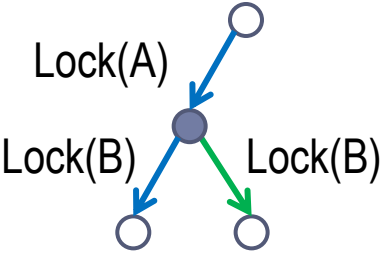
### Thread T1

```
→ 11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

### Thread T2

```
21: A.withdraw(50);
```

**B = 100**



```
1: class Account {
2:   Integer B = 100;
→ 3:   synchronized(B) getBalance() {
4:     return B;
5:   }
→ 6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```

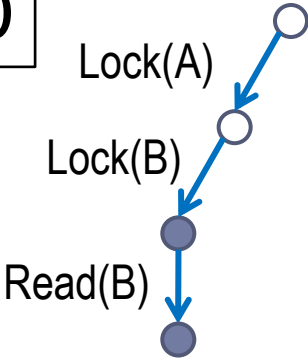
**Thread T1**

```
11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```

**B = 100**



```
1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }
```



**Thread T1**

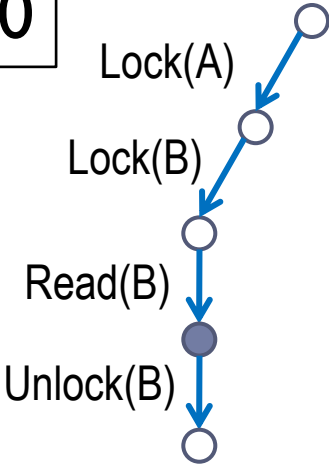
```
11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```

**B = 100**

```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
6:   synchronized(B) withdraw(int amt) {  
7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```



**Thread T1**

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```

**B = 100**

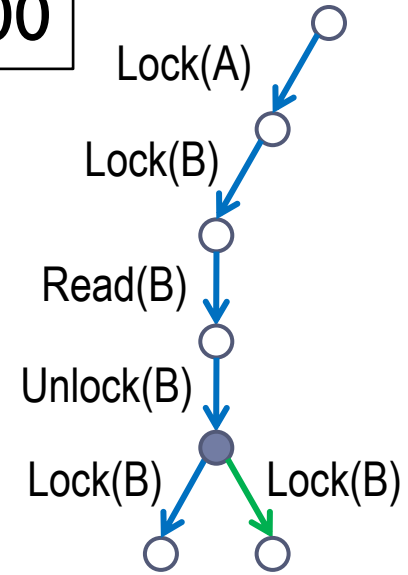
```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
➔ 6:   synchronized(B) withdraw(int amt) {  
7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

**Thread T1**

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```



**B = 100**

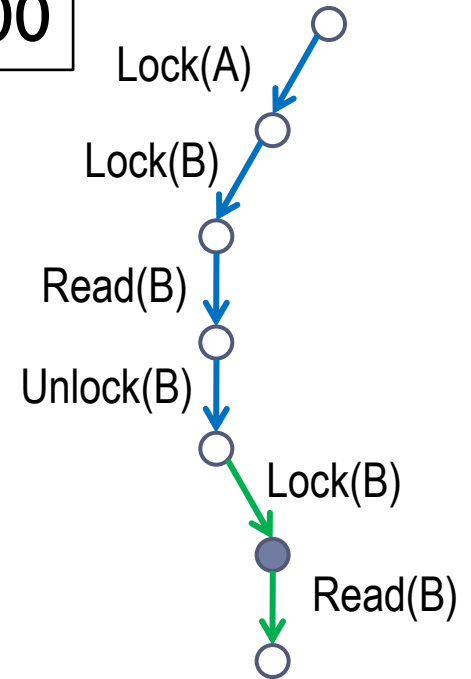
```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
→ 6:   synchronized(B) withdraw(int amt) {  
→ 7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

**Thread T1**

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```



**B = 100**

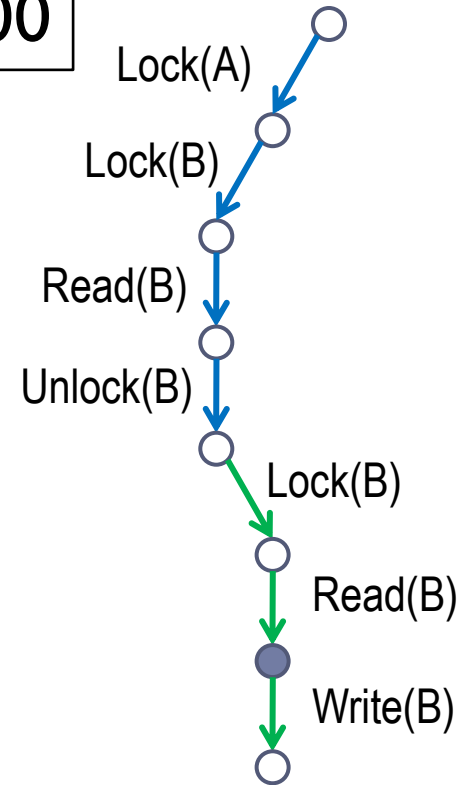
```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
→ 6:   synchronized(B) withdraw(int amt) {  
→ 7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

**Thread T1**

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```



**B = 50**

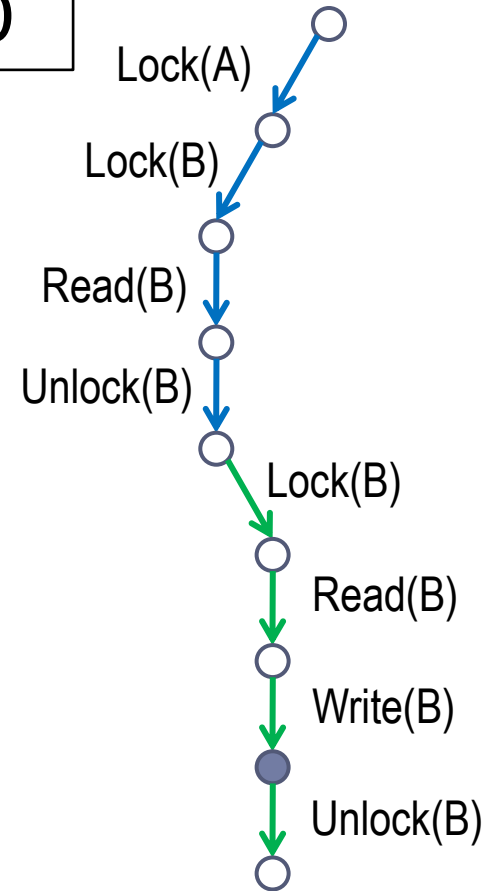
```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
→ 6:   synchronized(B) withdraw(int amt) {  
7:     B -= amt;  
8:     assert (B >= 0);  
→ 9:   }  
10: }
```

**Thread T1**

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```



**B = 50**

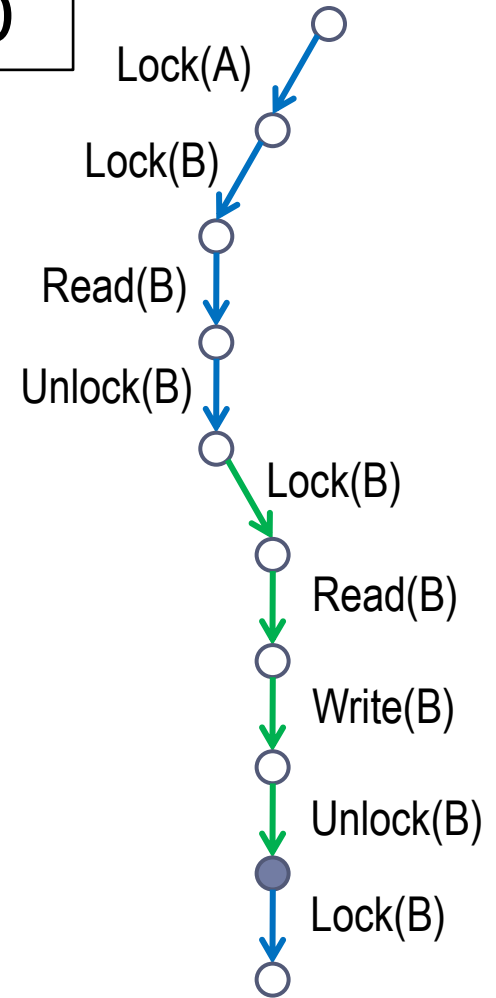
```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
→ 6:   synchronized(B) withdraw(int amt) {  
7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

**Thread T1**

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```



**B = 50**

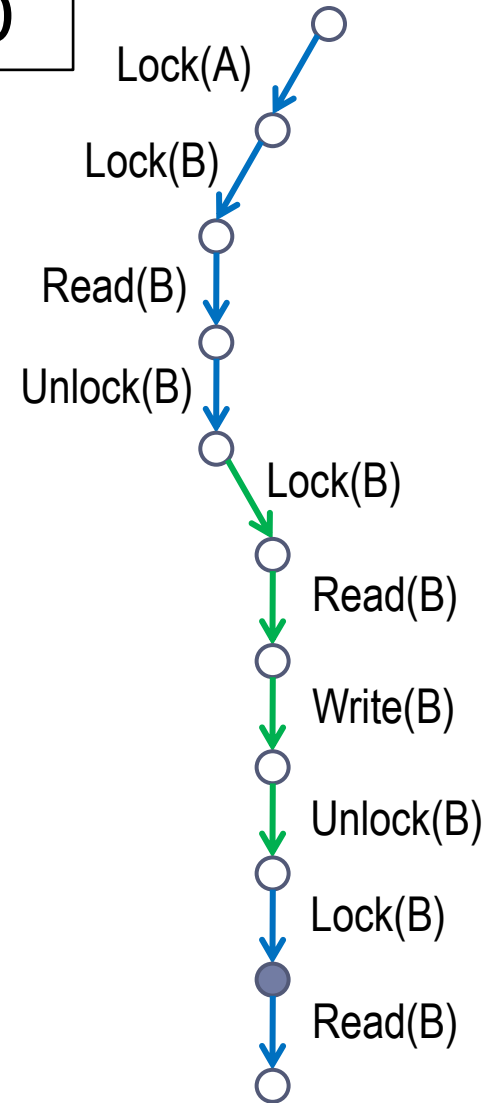
```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
6:   synchronized(B) withdraw(int amt) {  
→ 7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

**Thread T1**

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```



**B = 50**

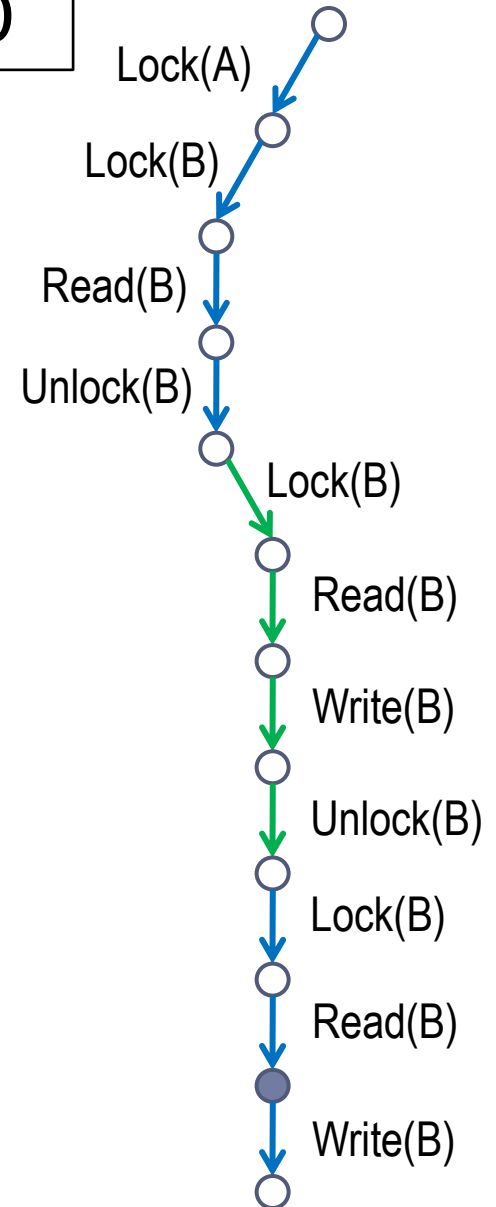
```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
6:   synchronized(B) withdraw(int amt) {  
→ 7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

**Thread T1**

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```



**B = -20**

```
1: class Account {  
2:   Integer B = 100;  
3:   synchronized(B) getBalance() {  
4:     return B;  
5:   }  
6:   synchronized(B) withdraw(int amt) {  
7:     B -= amt;  
8:     assert (B >= 0);  
9:   }  
10: }
```

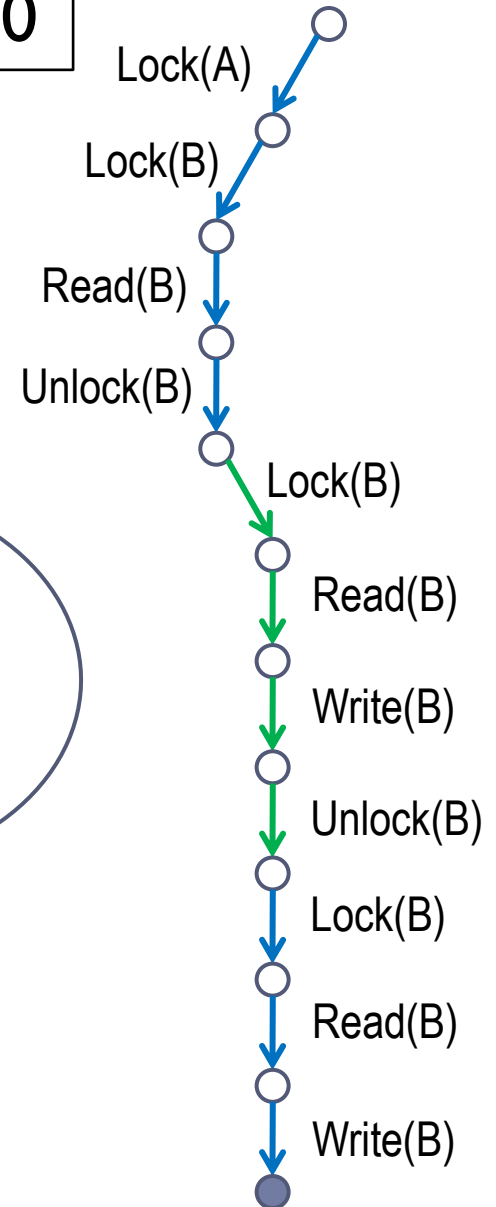


**Thread T1**

```
11: synchronized(A) { // @atomic  
12:   if(A.getBalance() >= 70)  
13:     A.withdraw(70);  
14: }
```

**Thread T2**

```
21: A.withdraw(50);
```



# Causal Atomicity

---

- ▶ **Dependency between transitions**
  - ▶ Transition from same thread
  - ▶ Acquire or release of same lock
  - ▶ Two accesses (at least one write) to same memory location
- ▶ **Happens-Before ( $\blacktriangleleft$ )**
  - ▶ If  $t_1$  is dependent on  $t_2$  and  $t_1$  executes before  $t_2$ ,  $t_1 \blacktriangleleft t_2$
- ▶ **Causal Atomicity Violation**
  - ▶  $t_1, t_3$  in same atomic block
  - ▶  $t_2$  from another thread
  - ▶  $t_1 \blacktriangleleft t_2$  and  $t_2 \blacktriangleleft t_3$

# Causal Atomicity

---

- ▶ **Dependency between transitions**
  - ▶ Transition from same thread
  - ▶ Acquire or release of same lock
  - ~~▶ Two accesses (at least one write) to same memory location~~
- ▶ **Happens-Before ( $\blacktriangleleft$ )**
  - ▶ If  $t_1$  is dependent on  $t_2$  and  $t_1$  executes before  $t_2$ ,  $t_1 \blacktriangleleft t_2$
- ▶ **Causal Atomicity Violation**
  - ▶  $t_1, t_3$  in same atomic block
  - ▶  $t_2$  from another thread
  - ▶  $t_1 \blacktriangleleft t_2$  and  $t_2 \blacktriangleleft t_3$

# A Pattern for Atomicity Violations

---

- ▶ **While inside an atomic block**
  - ▶ Acquire and release a lock
  - ▶ Before exiting the atomic block, reacquire the same lock
  - ▶ If some other thread can acquire same lock in between
    - ▶ This is a **causal atomicity violation**
- ▶ **Java does not have atomic sections**
  - ▶ For un-annotated code, use heuristic that synchronized methods denote atomic sections

```

1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```

### Thread T1

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

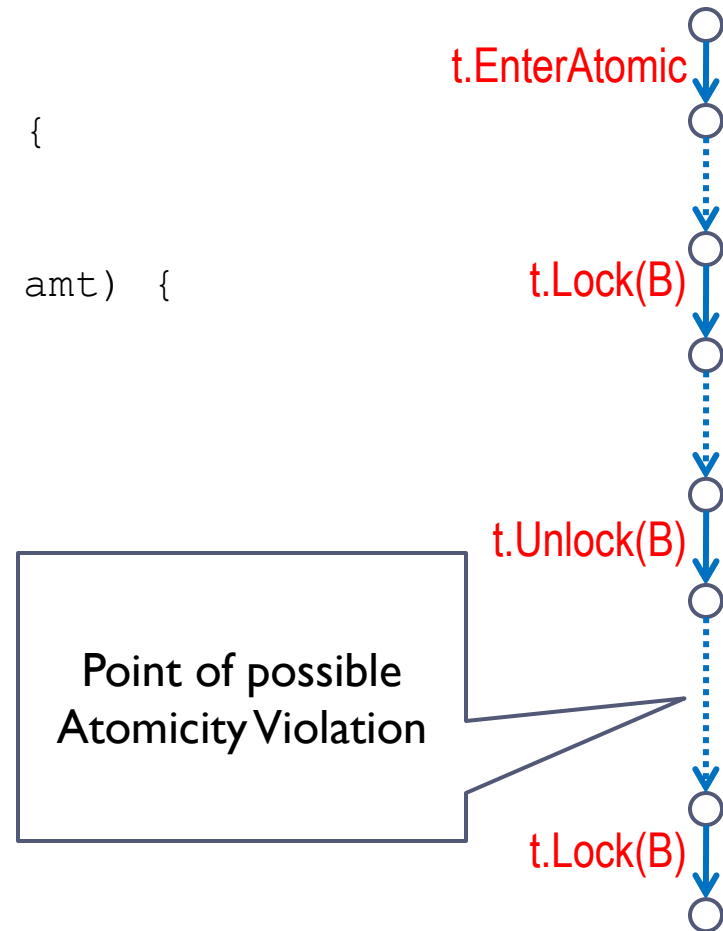
```

### Thread T2

```

21: A.withdraw(50);

```



# Randomized Active Atomicity Violation Detection

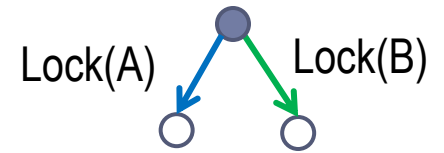
---

- ▶ **When inside an atomic block**
  - ▶ Keep track of locks acquired and released
- ▶ **When acquiring a lock**
  - ▶ If lock has been acquired and released in same atomic block, issue warning and pause thread with probability  $p$
  - ▶ If another thread is paused on lock, report atomicity violation

```

1: class Account {
2:     Integer B = 100;
3:     synchronized(B) getBalance() {
4:         return B;
5:     }
→ 6:     synchronized(B) withdraw(int amt) {
7:         B -= amt;
8:         assert (B >= 0);
9:     }
10: }

```



### Thread T1

```

→ 11: synchronized(A) { // @atomic
12:     if(A.getBalance() >= 70)
13:         A.withdraw(70);
14: }

```

### Thread T2

```

21: A.withdraw(50);

```

InsideAtomic



AlreadyAquired



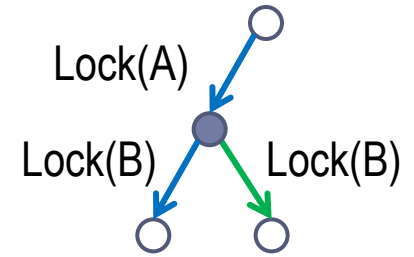
Paused



```

1: class Account {
2:   Integer B = 100;
→ 3:   synchronized(B) getBalance() {
4:     return B;
5:   }
→ 6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```



### Thread T1

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

```

### Thread T2

```

21: A.withdraw(50);

```

InsideAtomic

{T1}

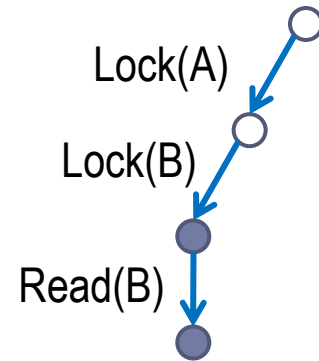
AlreadyAquired

Paused

```

1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```



### Thread T1

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

```

### Thread T2

```

21: A.withdraw(50);

```

InsideAtomic

{T1}

AlreadyAquired

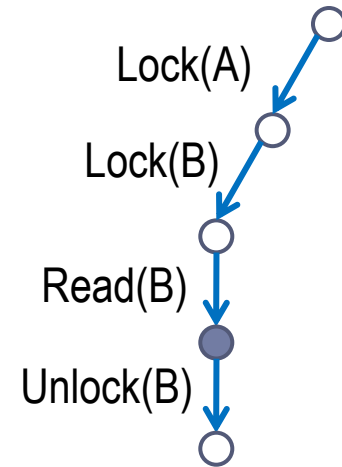
{(T1, B)}

Paused

```

1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```



**Thread T1**

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

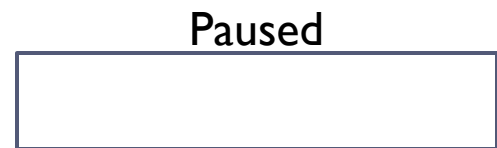
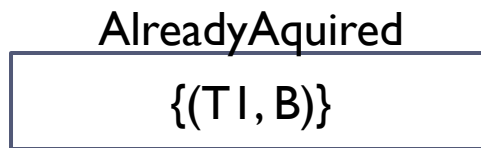
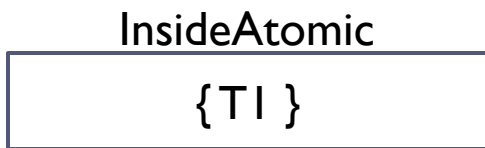
```

**Thread T2**

```

21: A.withdraw(50);

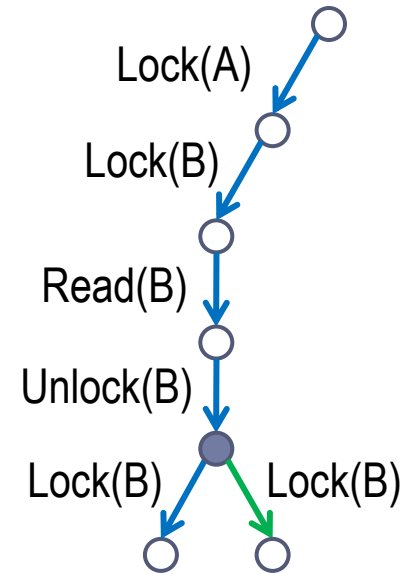
```



```

1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```



**Thread T1**

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

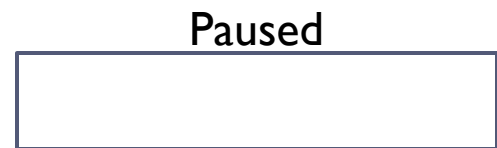
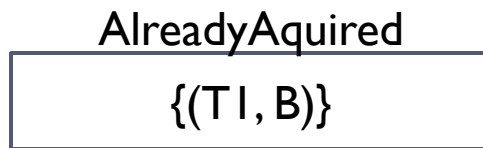
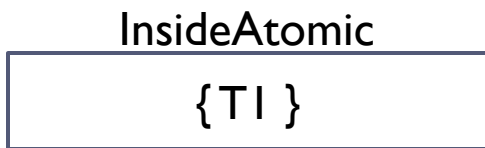
```

**Thread T2**

```

21: A.withdraw(50);

```



```

1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```



**Thread T1**

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

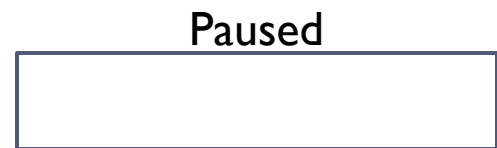
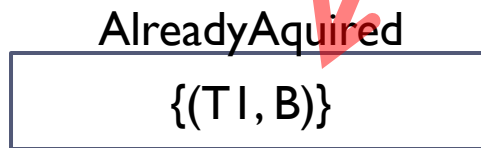
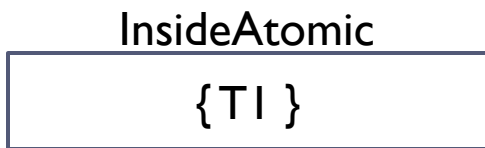
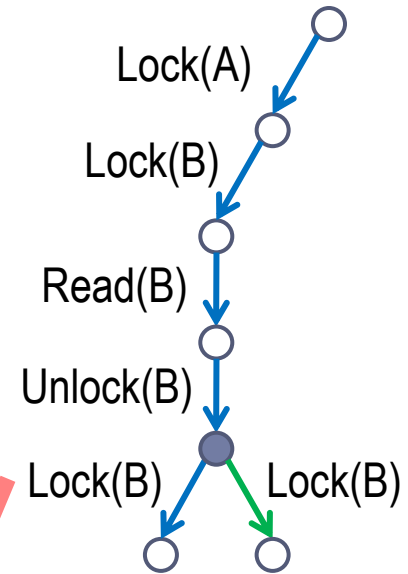
```

**Thread T2**

```

21: A.withdraw(50);

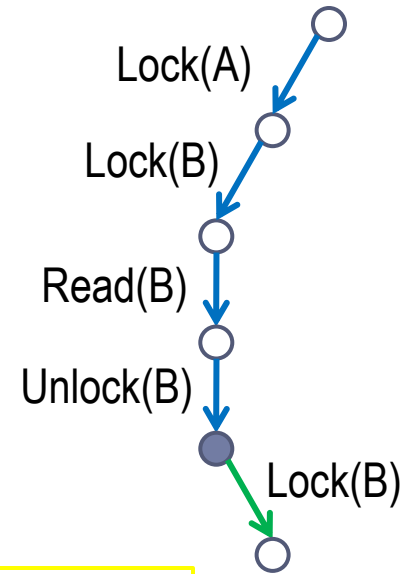
```



```

1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```



**Thread T1**

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

```

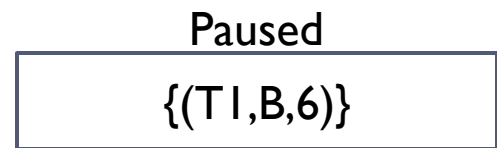
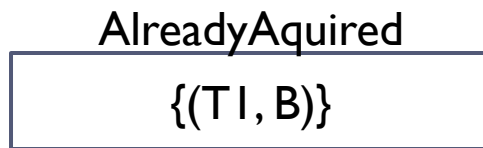
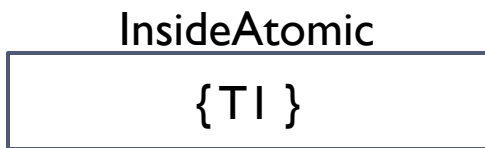
Atomicity Warning at T1,6

**Thread T2**

```

21: A.withdraw(50);

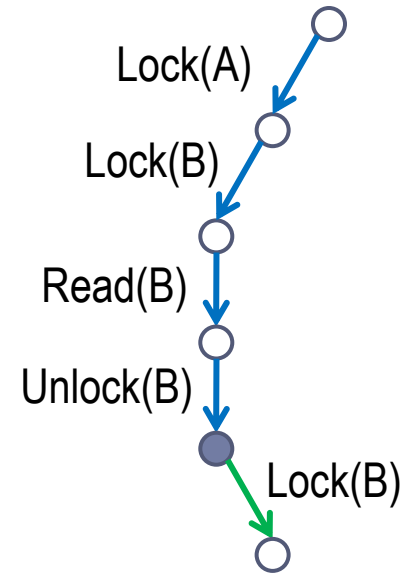
```



```

1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```



**Thread T1**

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

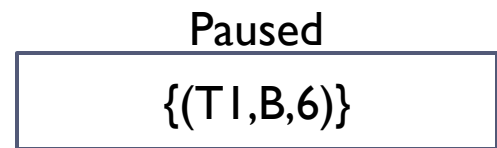
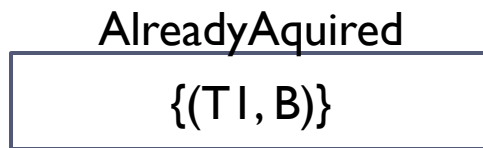
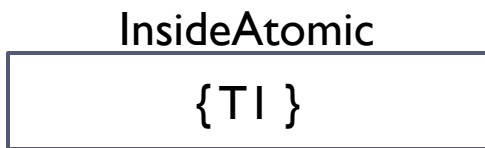
```

**Thread T2**

```

21: A.withdraw(50);

```



```

1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```

**Thread T1**

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

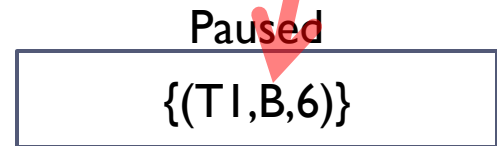
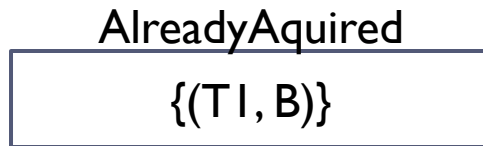
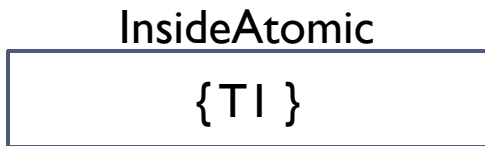
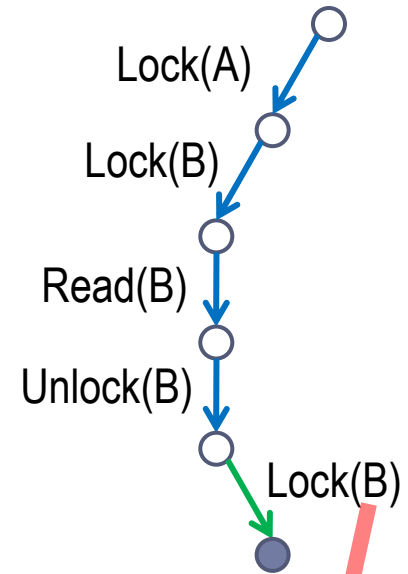
```

**Thread T2**

```

21: A.withdraw(50);

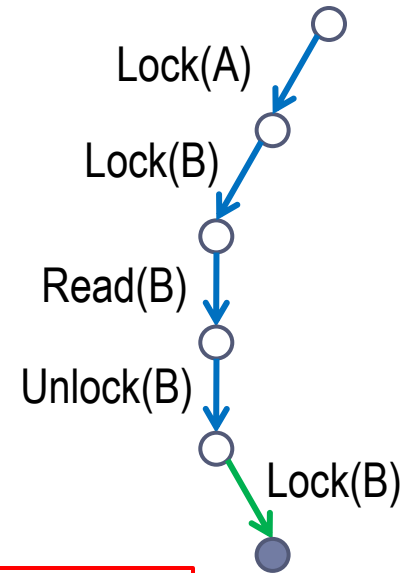
```



```

1: class Account {
2:   Integer B = 100;
3:   synchronized(B) getBalance() {
4:     return B;
5:   }
6:   synchronized(B) withdraw(int amt) {
7:     B -= amt;
8:     assert (B >= 0);
9:   }
10: }

```



**Thread T1**

```

11: synchronized(A) { // @atomic
12:   if(A.getBalance() >= 70)
13:     A.withdraw(70);
14: }

```

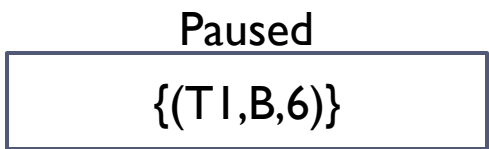
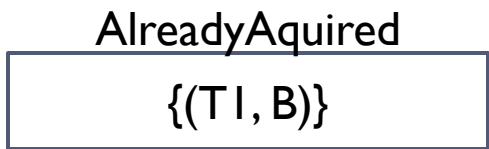
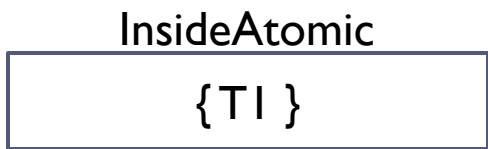
**Thread T2**

```

21: A.withdraw(50);

```

Atomicity Error between T1,6 and T2,6



# AtomFuzzer

---

- ▶ **Implementation in Java**
  - ▶ Instrument byte code to call our scheduler at
    - ▶ Atomic Enter
    - ▶ Atomic Exit
    - ▶ Lock Acquire
    - ▶ Lock Release
- ▶ **Used heuristics to infer atomic blocks for benchmarks without annotations**
- ▶ **Issues following reports:**
  - ▶ Warnings
  - ▶ Errors
  - ▶ Real Bugs

# Results

Program Name	LOC	Average Runtime	Slowdown	# Warnings	# Errors	# Bugs Confirmed	Prob. of Detection
StringBuffer	1,320	0.21 (s)	1.23	1	1	1	0.78
ArrayList	5,866	0.24	1.71	1	1	1	0.97
Apache BlockingBuffer	977	0.34	2.36	1	1	1	0.64
Google Concurrent Multiset	17,946	0.82	6.31	0	0	0	-
cache4j	3,897	35	10.6	1	1	0	1.00
hedc	29,947	1.8	1.82	3	0	0	-
weblech	35,175	13.78	17.01	25	0	0	-
jspider	64,933	51	10.6	28	4	0	1.00

# Example Bug (JDK 1.6)

---

```
class SynchronizedCollection {
    private Collection c1;
    private Object mutex;
    public boolean removeAll(Collection c2) {
L1 → synchronized(mutex) { return c1.removeAll(c2); }
    }
    public boolean contains(Object o) {
L2 → synchronized(mutex) {return c1.contains(o);}
    }
}
class AbstractCollection implements Collection {
    public boolean removeAll(Collection c2) {
        boolean modified = false;
        Iterator e = iterator();
        while (e.hasNext()) {
            [ if(c2.contains(e.next())) {
                e.remove();
                modified = true;
            }
        }
        return modified;
    }
}
```

# Limitations

---

- ▶ **Cannot detect all atomicity violations**
  - ▶ Violation has to be produced in the execution
    - ⇒ Combine with symbolic execution
  - ▶ Violation can be missed due to randomness
    - ⇒ Run multiple times
  - ▶ Misses violations that do not fit the pattern
    - ⇒ Use a datarace detector

# Sources of False Positives

- ▶ Heuristics to infer atomic blocks may not match programmer intention
  - ▶ synchronized run() methods
  - ▶ wait() inside a synchronized block
- ▶ Benign atomicity violations
  - ▶ Unique Ids
  - ▶ Counters for statistics

```
net.sf.cache4j
```

```
class BlockingCache {
```

```
2505: synchronized void incrementHits() {  
    hits++;  
}}
```

```
net.javacoding.jspider.core.threading
```

```
class WorkerThread {  
net.javacoding.jspider.core.threading  
116: public synchronized void run()  
class WorkerThreadPool {
```

```
42:     synchronized(this) {
```

```
        ..  
        ..
```

```
45:     wait();
```

```
        ..  
        ..
```

```
    hits++;  
}
```

# Conclusion

---

- ▶ Practical and effective technique to detect real atomicity violations in concurrent programs with high probability
- ▶ No false warnings with proper annotations
- ▶ Replay of buggy execution
- ▶ Detected previously unknown atomicity violations in JDK and Apache Commons framework

