Find Bugs in Concurrent Programs

Ao (Leo) Li

aoli@cs.cmu.edu

August 2025

Who am I

- Ao (Leo) Li
- PhD candidate at S3D/SCS advised by Rohan Padhye and Vyas Sekar
- Make complex concurrent and distributed systems easier to **debug** and **test**. I focus on both **algorithmic efficiency** and **real-world practicality**.

Ao (Leo) Li

antithesis Google Microsoft amazon

Will this program crash?

```
static int balance = 100;
```

```
void withdraw(int amount) {
  if (balance >= amount) {
    balance -= amount;
  }
}
```

```
void main() {
  Thread t1 = new Thread(() -> withdraw(100));
  Thread t2 = new Thread(() -> withdraw(100));
  t1.start();
  t2.start();
  t1.join();
  t2.join();
  assert balance >= 0; // can this assert fail?
}
```

It depends because this is a **concurrent** program.

What is Concurrency?

- Multiple components (threads, processes, machines) executing simultaneously.
- Components interact through shared memory, message passing, etc.
- Non-determinism: the exact order of interactions is unpredictable.
- Hard to debug and test.

Concurrency Bugs and Race Conditions

- Definition: some undesirable behavior that only manifests under certain orderings of component interactions.
- Data Race[1]
- Atomicity Violation
- Order Violation
- Deadlock
- ...

1: Not everyone agrees that all data races are race conditions. See John Regehr's post: Race Condition vs. Data Race

Data Race (in Java)

A data race occurs when two threads access the same memory location concurrently, and at least one of the accesses is a write.

```
a = 0;
b = 0;
```

Thread 1:

```
a = 1;
r1 = b;
```

Thread 2:

```
b = 1;
r2 = a;
```

Possible outcomes: (r1, r2) = (1, 0), (0, 1), (0, 0), and (1, 1).

Why data races can be problematic?

Each core writes to its local cache/buffer before updating the main memory.

One thread may not see the most recent writes from another thread.

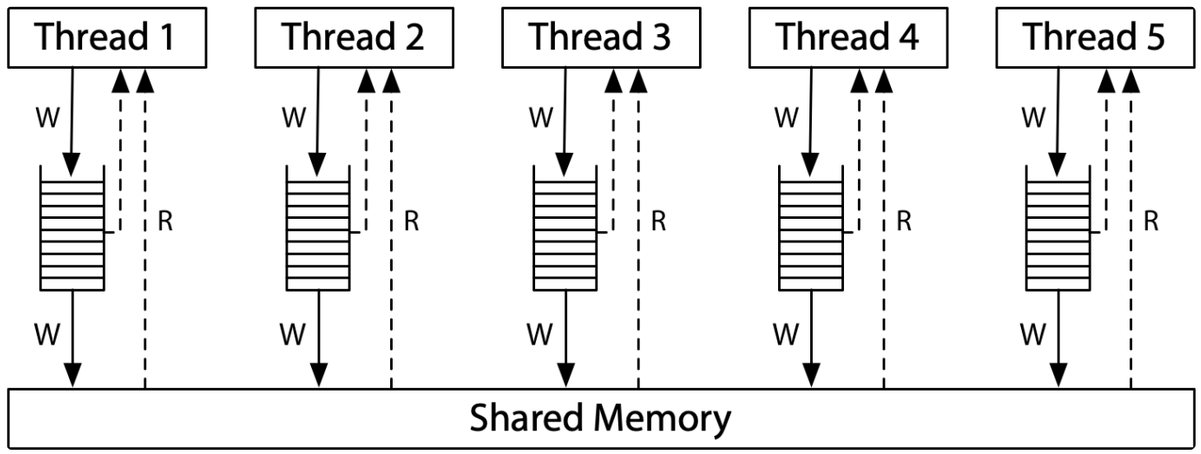


Figure from: https://research.swtch.com/hwmm.

Bonus: will data race happen on a single core machine?

Answer: Yes! https://aoli.al/blogs/data-race/

Why data races can be problematic? (Cont'd)

Data races

Different threads of execution are always allowed to access (read and modify) different memory locations concurrently, with no interference and no synchronization requirements.

Two expression evaluations *conflict* if one of them modifies a memory location or starts/ends the lifetime of an object in a memory location, and the other one reads or modifies the same memory location or starts/ends the lifetime of an object occupying storage that overlaps with the memory location.

A program that has two conflicting evaluations has a *data race* unless

- both evaluations execute on the same thread or in the same signal handler, or
- both conflicting evaluations are atomic operations (see `std::atomic`), or
- one of the conflicting evaluations *happens-before* another (see `std::memory_order`).

If a data race occurs, the behavior of the program is undefined.

Example 17.4-1. Incorrectly Synchronized Programs May Exhibit Surprising Behavior

The semantics of the Java programming language allow compilers and microprocessors to perform optimizations behaviors.

Consider, for example, the example program traces shown in Table 17.4-A. This program uses local variables `r1`

Table 17.4-A. Surprising results caused by statement reordering - original code

How to Detect Data Races?

Thread 1:

```
1 memory_write_operation(Thread 1, &balance);
2 balance = 0;
```

Thread 2:

```
1 memory_read_operation(Thread 2, &balance);
2 if (balance > 100) {...}
```

- Get context for the memory location being accessed.
- Compare current thread's **vector clock** with last access.
  - Vector clock is a data structure to understand if two operations are ordered or concurrent.
- If there's a previous access from a different thread AND the vector clock show they are not ordered AND at least one is a write.
  - **Report a data race**

Time, Clocks, and the Ordering of Events in a Distributed System by Leslie Lamport: https://lamport.azurewebsites.net/pubs/time-clocks.pdf

How to Detect Data Races? (Cont'd)

- Static Solutions
  - OxCaml
  - Rust Borrow Checker
  - Infer (RacerD)
- Dynamic Solutions
  - ThreadSanitizer (TSan)
  - Go Race Detector
  - Helgrind (Valgrind)

How to Write Data Race Free Code?

- Use (safe) Rust 🦀

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 ⟶ rust-borrow-checker.rs:4:14
  |
3 |     let r1 = &s;
  |              -- immutable borrow occurs here
4 |     let r2 = &mut s;
  |              ^^^^^^ mutable borrow occurs here
5 |     println!("{}, and {}", r1, r2);
  |                            -- immutable borrow later used here
```

- Use atomic operations.

```
static AtomicInteger balance = 100;
```

*This is simplified syntax.

- Use locks.

```
lock.lock();
if (balance >= amount) {...}
lock.unlock();
```

Data Race (Revisit)

```
a = 0;
b = 0;
```

Thread 1:

```
r1 = a;
r2 = a;
if (r1 == r2) {
  b = 2;
}
```

Thread 2:

```
r3 = b;
a = r3
```

Will r1 = r2 = r3 = 2 be possible?

Yes

Data Race (Revisit)

```
a = 0;
b = 0;
```

Thread 1:

```
r1 = a;
if (r1 != 0) {
  b = 42;
}
```

Thread 2:

```
r2 = b;
if (r2 != 0) {
  a = 42;
}
```

Will r1 = r2 = 42 be possible?

No, but why?

The Java Memory Model

Defines how threads in a Java program interact through memory and what behaviors are allowed in concurrent execution.

- Each read can only see writes that happen before them. Reads cannot see writes through data races.
- Synchronization order is consistent with program order and mutual exclusion.
- The execution obeys intra-thread consistency.
- The execution obeys synchronization-order consistency. (A read cannot see a write that is "synchronized" after it.)
- The execution obeys happens-before consistency. (A read cannot see a write that happens after it.)

Litmus Testing

Small tests to verify specification behaviors or potential bugs.

```
litmusTest ({
  object : LitmusIIOutcome () { var x , y = 0, 0 }
}) {
  thread { x = 1; r1 = y }
  thread { y = 1; r2 = x }
  spec {
    accept (0 , 1)
    accept (1 , 0)
    accept (1 , 1)
    interesting (0 , 0)
  }
}
```

- Accept: sequentially consistent outcomes
- Interesting: weak but tolerable outcomes
- Forbidden: weak intolerable outcomes

Litmus Testing is Stress Testing

- Run the litmus test many times (millions or more).
- Run the same tests on different cores.
- Use different compiler optimizations.

- LitmusKt: Concurrency Stress Testing for Kotlin
- Java Concurrency Stress (jcstress)
- ...

Concurrency Bugs and Race Conditions

- Definition: some undesirable behavior that only manifests under certain orderings of component interactions.
- Data Race ✅
- Atomicity Violation
- Order Violation
- Deadlock
- ...

Atomicity Violation

```
static AtomicInteger balance = 100;
void withdraw(int amount) {
  if (balance >= amount) {
    balance -= amount;
  }
}
```

Two threads call `withdraw(100)`:

Thread 1	Thread 2	Shared balance
`r1 = load(balance) // 100`	–	100
–	`r2 = load(balance) // 100`	100
`if (r1 >= 100) store(balance, 100 - 100)`	–	**0**
–	`if (r2 >= 100) store(balance, 0 - 100)`	**-100**

The withdraw operation is not atomic.

The **desired serializability** among **multiple memory accesses** is violated.

Atomicity Violation: the Fix

```
1 static AtomicInteger balance = 100;
2 static ReentrantLock lock = new ReentrantLock();
3 void withdraw(int amount) {
4   lock.lock();
5   if (balance >= amount) {
6     balance -= amount;
7   }
8   lock.unlock();
9 }
```

Why don't we put locks everywhere?

Performance overhead; deadlocks; ...

Order Violation

```
void fastWithdraw(int amount) {
  int oldBalance = balance;
  new Thread(() -> withdraw(amount/2)).start();
  new Thread(() -> withdraw(amount - amount/2)).start();
  assert(balance == oldBalance - amount); // can this assert fail?
}
```

Main thread	Thread A	Thread B
old = balance // 100	–	–
start(A)	ready	–
start(B)	ready	ready
assert(balance == old - amount)	– (not run yet)	– (not run yet)
–	withdraw(amount/2) // balance: 50	–
–	–	withdraw(amount-amount/2) // balance: 0

The assert may observe `balance` as **100** (or **50**).

The **desired order** between two (groups of) memory accesses is **flipped**.

Order Violation: the Fix

```
1  void fastWithdraw(int amount) {
2    int oldBalance = balance;
3    Thread t1 = new Thread(() -> withdraw(amount/2));
4    t1.start();
5    t1.join(); // Blocked until t1 finishes
6    Thread t2 = new Thread(() -> withdraw(amount - amount/2));
7    t2.start();
8    t2.join(); // Blocked until t2 finishes
9    assert(balance == oldBalance - amount); // can this assert fail?
10 }
```

Better fix? Did we introduce unnecessary ordering?

Atomicity/Order Violation v.s. Data Race

- Data race
  - Two memory operations ... same time ... one is write
  - Objective (not always a bug)
  - Can be detected through language/runtime checks
- Atomicity/order violation
  - Desired order/serializability ... is violated
  - Subjective (always a bug)
  - Detection depends on application requirements/specifications

Learning from mistakes: a comprehensive study on real world concurrency bug characteristics Lu et al. ASPLOS '08

How to Find Atomicity/Order Violation Bugs?

```
void fastWithdraw(int amount) {
  int oldBalance = balance;
  new Thread(() -> withdraw(amount/2)).start();
  new Thread(() -> withdraw(amount - amount/2)).start();
  assert(balance == oldBalance - amount); // can this assert fail?
}
```

Some possible interleavings

- `withdraw(amount/2)` → `withdraw(amount - amount/2)` → `assert`
- `withdraw(amount - amount/2)` → `withdraw(amount/2)` → `assert`
- `assert` → `withdraw(amount/2)` → `withdraw(amount - amount/2)`
- `withdraw(amount/2)` → `assert` → `withdraw(amount - amount/2)`

Concurrency testing: test a concurrent program under different interleavings.

Approach 1: Simple/Stress Testing

```
void fastWithdrawTest() {
  for (int i = 0; i < 1000; i++) {
    balance = 100;
    fastWithdraw(100);
  }
}
```

- Run the program many times.
- Hope to hit the buggy interleaving.
- Pros:
  - Easy to implement; no modification to the program; ...
- Cons:
  - Non-deterministic; not efficient; ...

Approach 2: Directed Yielding

```
1 void fastWithdraw(int amount) {
2   int oldBalance = balance;
3   new Thread(() -> withdraw(amount/2)).start();
4   Thread.sleep(1000);
5   new Thread(() -> withdraw(amount - amount/2)).start();
6   assert(balance == oldBalance - amount); // can this assert fail?
7 }
```

- Insert sleep/locks to enforce certain interleavings.
- IMUnit, ThreadWeaver, CalFuzzer, ...
- Pros:
  - Good for experts who know what interleavings to test
- Cons:
  - Spurious deadlocks; manual efforts

Approach 3: Controlled Concurrency Testing

- Systematically explore different interleavings.
- Shuttle (Rust), Coyote (C#), Fray (JVM), ...
- Pros:
  - Good coverage; can find deep bugs; ...
- Cons:
  - High overhead; hard to implement; ...

Fray: CCT for the JVM

- Deterministic controlled concurrency testing.
- Easy to use and general purpose.
- Support various concurrency testing algorithms.
- Gradle/Maven plugin + Jetbrains debugger.
- Core Idea: Sequential Execution + Control
- https://github.com/cmu-pasta/fray

```
Thread 1:

r1=AtomicRead(b)
if(r1>=amount)
 r2=AtomicRead(b)
 AtomicWrite(b, r2-a)
endif
```

```
void withdraw(int amount) {
  if (balance >= a) {
    b -= a;
  }
}
```

```
Thread 1:

🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

```
Thread 1:

r1=AtomicRead(b)
if(r1>=amount)
 r2=AtomicRead(b)
 AtomicWrite(b, r2-a)
endif
```

```
Thread 2:

r1=AtomicRead(b)
if(r1>=amount)
 r2=AtomicRead(b)
 AtomicWrite(b, r2-a)
endif
```

```
Thread 1:

🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

```
Thread 2:

🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
   🔒2[b].lock()
   r1=AtomicRead(b)
   🔒2[b].unlock();
```

Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

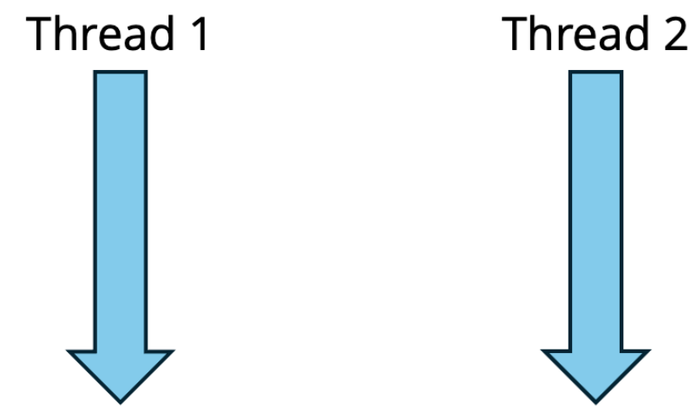
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
   🔒2[b].lock()
   r1=AtomicRead(b)
   🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by Fray

🔒2[b]: Held by Fray



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

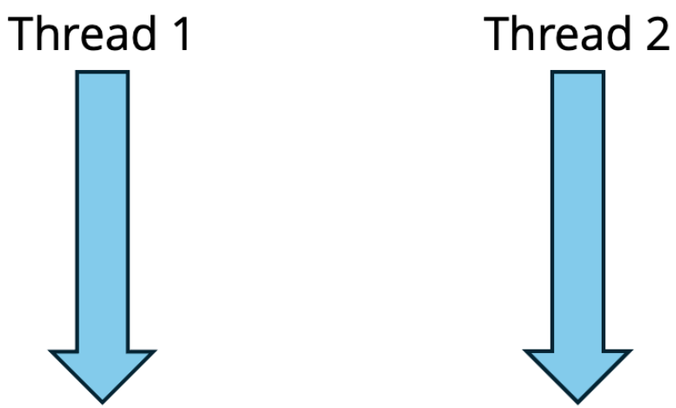
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
   🔒2[b].lock()
   r1=AtomicRead(b)
   🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Available

🔒2[b]: Held by Fray



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

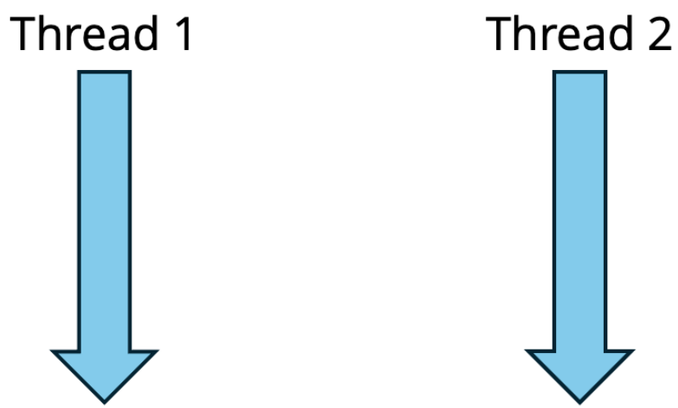
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
   🔒2[b].lock()
   r1=AtomicRead(b)
   🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by T1

🔒2[b]: Held by Fray



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

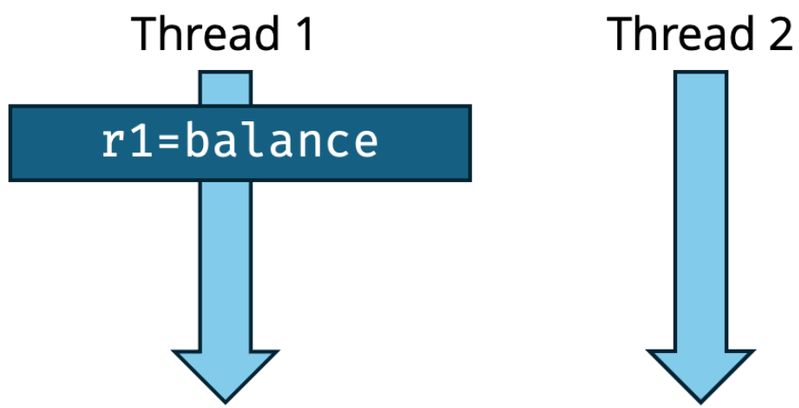
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
   🔒2[b].lock()
   r1=AtomicRead(b)
   🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by Fray

🔒2[b]: Held by T1



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
    🔒1[b].lock()
    r1=AtomicRead(b)
    🔒1[b].unlock();
```

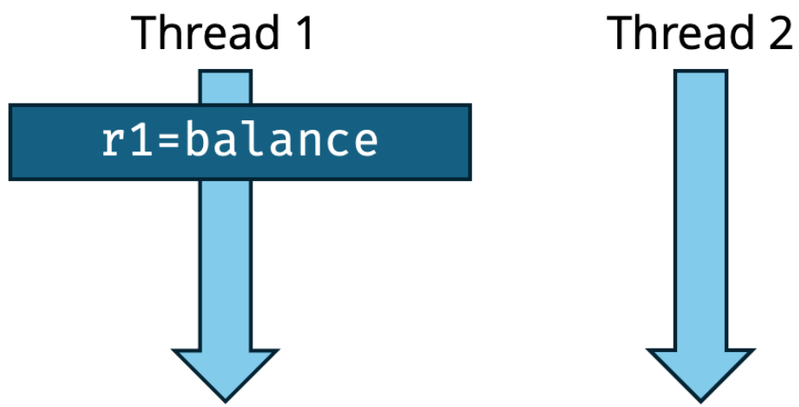
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
    🔒2[b].lock()
    r1=AtomicRead(b)
    🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by Fray

🔒2[b]: Held by Fray



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

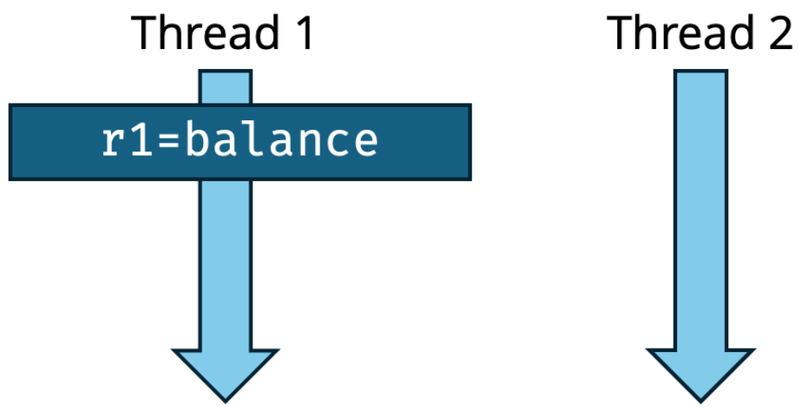
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
   🔒2[b].lock()
   r1=AtomicRead(b)
   🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by Fray

🔒2[b]: Held by Fray



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

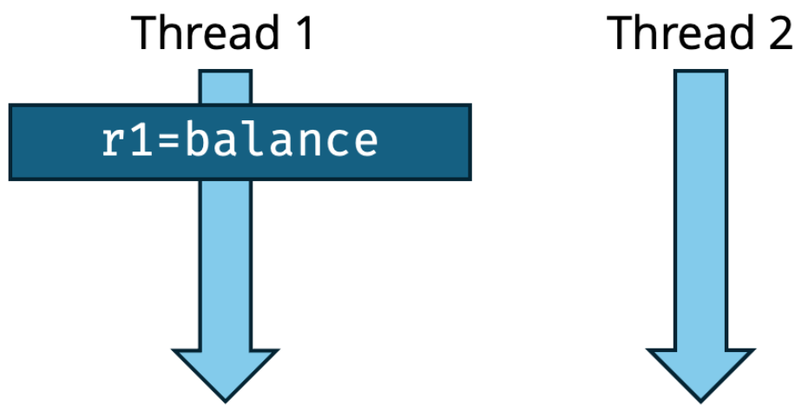
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
   🔒2[b].lock()
   r1=AtomicRead(b)
   🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by Fray

🔒2[b]: Available



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

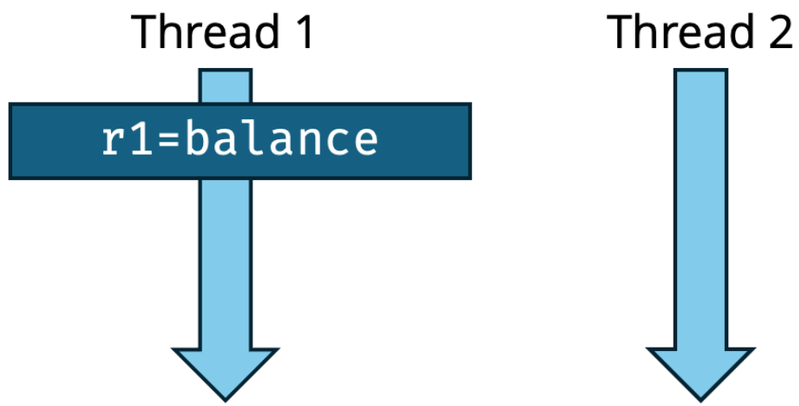
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
   🔒2[b].lock()
   r1=AtomicRead(b)
   🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by Fray

🔒2[b]: Held by T2



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
   🔒1[b].lock()
   r1=AtomicRead(b)
   🔒1[b].unlock();
```

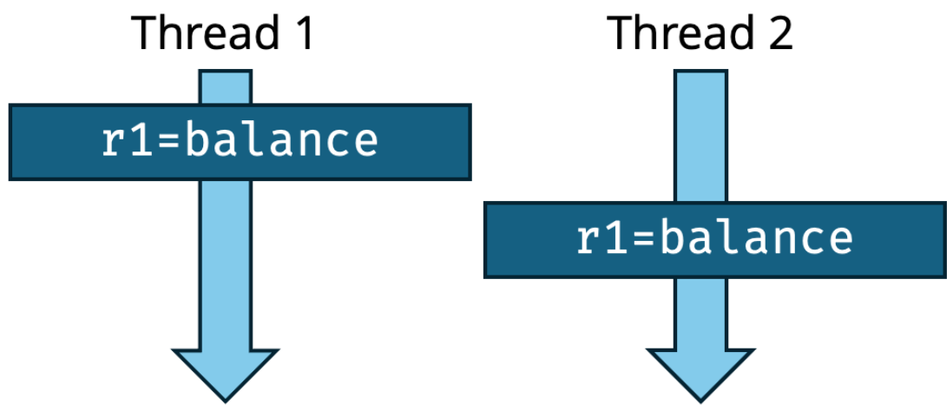
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
   🔒2[b].lock()
   r1=AtomicRead(b)
   🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by Fray

🔒2[b]: Held by T2



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
    🔒1[b].lock()
    r1=AtomicRead(b)
    🔒1[b].unlock();
```

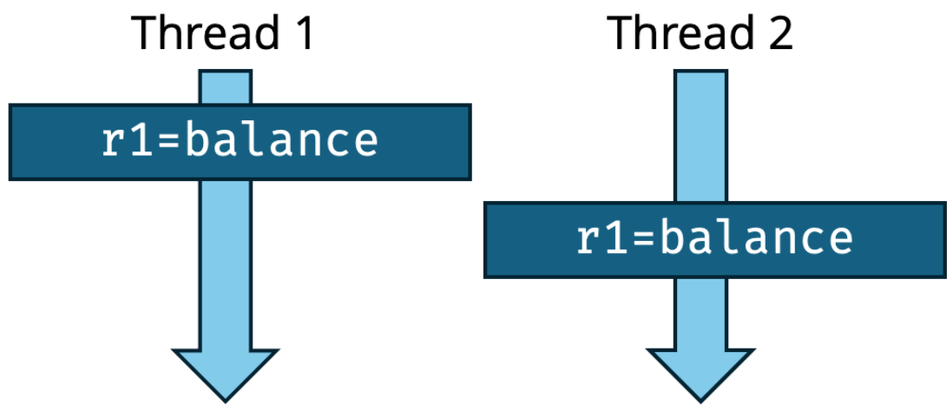
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
    🔒2[b].lock()
    r1=AtomicRead(b)
    🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by Fray

🔒2[b]: Held by Fray



Thread 1:

```
🔒1[b].lock()
r1=AtomicRead(b)
🔒1[b].unlock();
if(r1>=amount)
    🔒1[b].lock()
    r1=AtomicRead(b)
    🔒1[b].unlock();
```

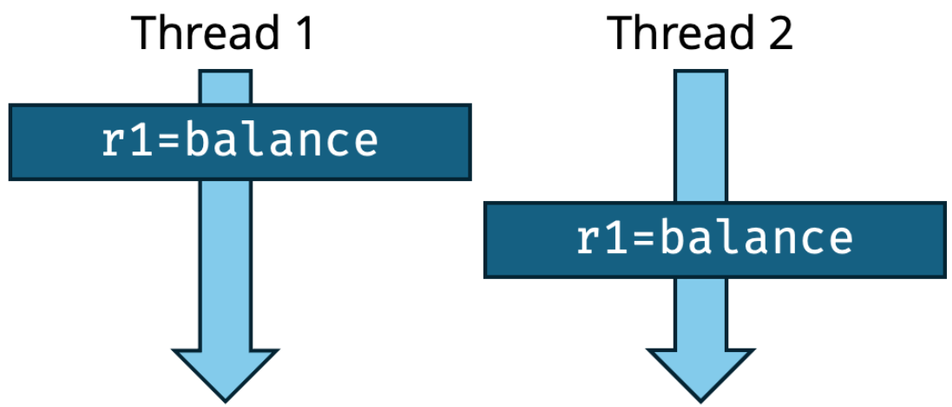
Thread 2:

```
🔒2[b].lock()
r1=AtomicRead(b)
🔒2[b].unlock();
if(r1>=amount)
    🔒2[b].lock()
    r1=AtomicRead(b)
    🔒2[b].unlock();
```

Lock Status:

🔒1[b]: Held by Fray

🔒2[b]: Held by Fray



Concurrency Testing Algorithms

- Question: what thread interleaving shall we test?
- Sol1. Exhaustive Testing
- Sol2. Random Walk
- What will you do?

Probabilistic Concurrency Testing (PCT)

- Empirical Observation: many concurrency bugs can be triggered with only **small number** of context switches (bug depth).
  - A context switch is when program execution switches from one thread to another.
- Idea: only explore interleavings with small number of context switches.

A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs

Partial Order Aware Conc. Testing (POS)

Thread 1:

```
balance = 100;
```

Thread 2:

```
total_account = 1;
```

- 2 possible interleavings:
  - `balance=100` → `total_account=1`
  - `total_account=1` → `balance=100`
- Do we need to explore all of them?
- Idea: only explore interleavings that are **semantically non-equivalent**.

Partial Order Aware Concurrency Sampling

Other Concurrency Testing Algorithms (🔥)

- Greybox Fuzzing for Concurrency Testing (ASPLOS '24)

- Selectively Uniform Concurrency Testing (ASPLOS '25)

- Feedback-guided Adaptive Testing of Distributed Systems Designs (NSDI '26)

Want to Learn More?

- Distributed System Testing
  - Jepsen
  - Greybox Fuzzing of Distributed Systems CCS '23
- Model checking and verification
  - TLA+
  - P Framework
- Testing Distributed Systems

Find Bugs in Concurrent Programs

Ao (Leo) Li

aoli@cs.cmu.edu

October 2025