Lecture 21: Concurrency Bugs

Deadlocks, Data Races, Atomicity/Ordering Violations

Xin Liu

Florida State University
xliu15@fsu.edu

CIS 5370 Computer Security
https://xinliulab.github.io/cis5370.html

Outline

Observation

When we write programs, we are, in a sense, writing bugs.

Today's Key Question:

- Even today, we still lack effective and convenient techniques to help us quickly build reliable software systems, even with
  - Rust
  - JavaScript
- Concurrency bugs and the security issues they cause often have an elusive nature, which makes them prone to slipping past developers' control.

Main Topics for Today:

- Deadlocks
- Data Races
- Atomicity/Ordering Violations

Concurrency Bugs that Cost Lives

A Retrospective: Item Duplication in Diablo I

How to duplicate items in Diablo I

Event-Level Concurrency

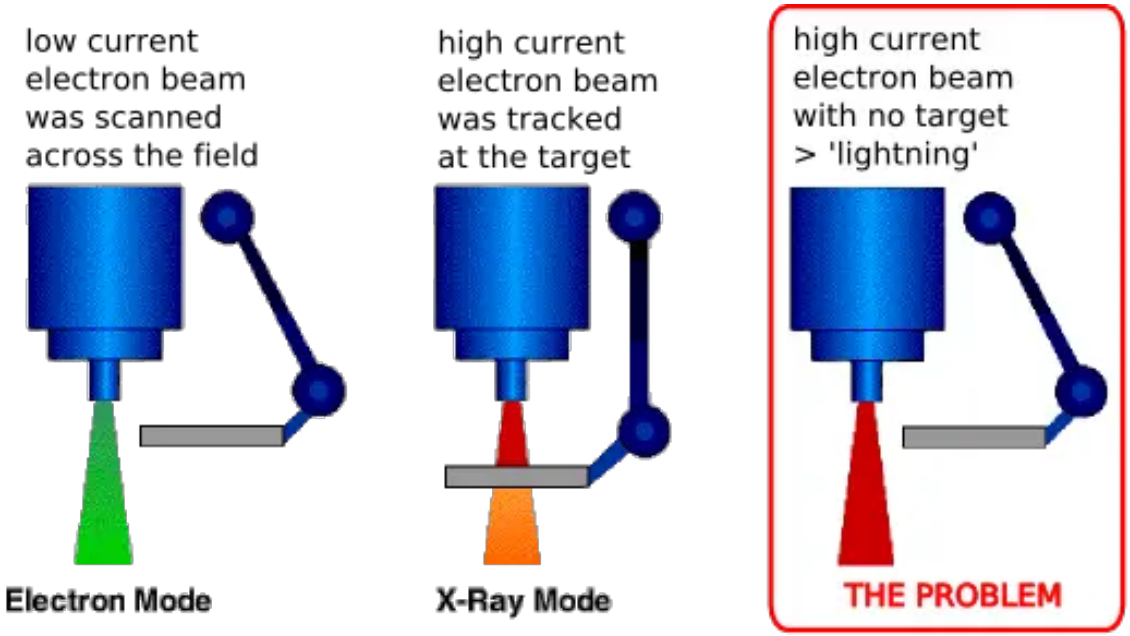
```
Event(doMouseMove) {
    hoveredItem = Item("$1");
}

// Unexpected interleaved event
Event(clickEvent) {
    hoveredItem = Item("$99"); // <- Shared state
    Inventory.append(hoveredItem);
}

Event(doPickUp) {
    InHand = hoveredItem;
}
```

Killed by a Machine

Therac-25 Incident (1985–1987)

- A concurrency bug triggered by event-driven programming caused by race conditions, leading to the deaths of at least 6 people.

The Therac-25



Safety Assertions (to prevent dangerous mode)

```
assert mode in [Electron(Low), XRay(High)]
assert mirror in [On, Off]
assert not (mode == XRay(High) and mirror == Off)
```

The Killer Software Bug in History

Diablo I Case Reproduction

- Select **X-Ray (High)** Mode
  - The machine starts moving the `mirror`, which takes about 8 seconds
- Switch to **Electron (Low)** Mode (OK)
- Quickly switch back to **X-Ray (High)** Mode
- Assertion fail: Malfunction 54; the operator continued `without` intended confirmation

The Tragedy of Full Digital Control

- In a newer product (Therac-20), `assertion fail` could trigger a circuit interlock bypass, directly activating the machine, requiring manual reset.

This Wasn't Even the Last Killer Bug in Therac-25

After the Fix...

- If the operator sent a command at the exact moment the counter overflowed, the machine would skip setting up some of the beam accessories.

Final Resolution

- Introduce an independent hardware safety mechanism
- Immediately halt machine when excessive radiation is detected

Reflection: Where Is the "Last Line of Defense" in the AI Era?

- Will we one day live in illusions generated by AI, without realizing it?

More Bizarre Bugs

Even more bizarre bugs have occurred throughout history — including severe incidents caused by concurrency. One notable example is the 2003 blackout in the U.S. and Canada, which was estimated to have caused economic losses of $25 to $30 billion.

Observation

One of the main reasons concurrency bugs are so elusive is their inherent non-determinism. Even after rigorous testing, rare scheduling interleavings can still trigger chain reactions. It wasn't until around 2010 that both academia and industry began to develop a systematic understanding of correctness in concurrent systems.

Deadlock

Deadlock

A deadlock is a state in which each member of a group is waiting for another member, including itself, to take action.

Deadlocks aren't just about multiple threads

```
// Good
lock(&lk);
// xchg(&lk->locked, LOCKED) == True
...

// Possibly in interrupt handler
lock(&lk);
// xchg(&lk->locked, LOCKED) == False
```

Looks silly? Think you won't make this mistake?

- No — you will!
- Real-world systems have complexity waiting to trap you:
  - Deep call stacks
  - Hidden control flow

Multi-Thread Deadlock

Philosopher Dining Problem

```
void T_philosopher() {
    P(&avail[lhs]);
    P(&avail[rhs]);
    // ...
    V(&avail[lhs]);
    V(&avail[rhs]);
}
```

- T_1: P(0) – success, P(1) – waiting
- T_2: P(1) – success, P(2) – waiting
- T_3: P(2) – success, P(3) – waiting
- T_4: P(3) – success, P(4) – waiting
- T_5: P(4) – success, P(0) – waiting

Dining Philosophers Problem (E. W. Dijkstra, 1960)

- Philosophers (threads) alternate between thinking and eating
- Eating requires simultaneously picking up both the left and right forks
- When a fork is occupied by another philosopher, they must wait
- How to achieve synchronization?
  - Use mutexes or semaphores to implement synchronization
  - Ensure that only one philosopher can pick up both forks at a time

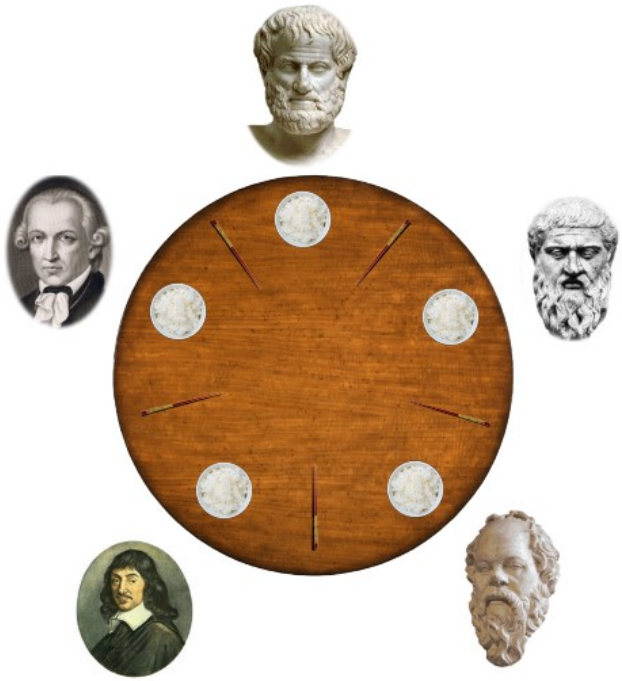


Illustration of the Dining Philosophers Problem

Failed and Successful Attempts

Failed Attempt:

- `philosopher.c` (How to solve?)

```
// Failed Attempt
void *philosopher(void *arg) {
   while (1) {
      // Thinking...
      pick_up_forks();
      // Eating...
      put_down_forks();
   }
}
```

Successful Attempt: The Universal Solution

- Use mutexes and condition variables for synchronization
- Ensure mutual exclusion when checking and updating fork availability

```
// Successful Attempt (Mutex-based)
mutex_lock(&mutex);
while (!(avail[lhs] && avail[rhs])) {
  wait(&cv, &mutex);
}
avail[lhs] = avail[rhs] = false;
mutex_unlock(&mutex);

mutex_lock(&mutex);
avail[lhs] = avail[rhs] = true;
broadcast(&cv);
mutex_unlock(&mutex);
```

Leader/Follower Solution: Centralized Management

Forget semaphores, let one person manage the forks!

- **Leader/follower** - Producer/Consumer model
- Common solution in distributed systems (e.g., HDFS, ...)

```
// Philosopher function
void Tphilosopher(int id) {
  send_request(id, EAT);
  P(allowed[id]); // waiter hands forks to
        philosopher
  philosopher_eat();
  send_request(id, DONE);
}
```

```
// Waiter function
void Twaiter() {
  while (1) {
    (id, status) = receive_request();
    if (status == EAT) { ... }
    if (status == DONE) { ... }
  }
}
```

Forget Complex Synchronization Algorithms!

- You might think that the person managing the forks is a performance bottleneck.
- Imagine a large table where everyone is calling the waiter at once.
- *"Premature optimization is the root of all evil"* (D. E. Knuth)
- Optimizing without understanding the workload is reckless.

Key Insight:

- Dining time is usually much longer than the time it takes to request the waiter.
- If one manager cannot handle it, you can split the workload (fast/slow path).
- Design the system so centralized management doesn't become a bottleneck.
- Reference: *Millions of tiny databases (NSDI'20)*.

Necessary Conditions for Deadlock

System Deadlocks (1971): Think of locks as balls inside bags

1. **Mutual Exclusion** – Only one thread can hold a ball (lock) at a time
2. **Wait-for** – A thread holding a ball may wait for more
3. **No Preemption** – Balls (locks) cannot be forcibly taken away
4. **Circular Chain** – A circular wait forms among threads holding and waiting for balls

"Necessary" Conditions?

- If you break *any one* of the four conditions, deadlock cannot occur.

Necessary Conditions for Deadlock (cont'd)

Understanding the causes of deadlock—especially the four necessary conditions—allows us to effectively avoid, prevent, and resolve deadlocks. Therefore, system design and process scheduling must consider how to avoid satisfying all four conditions, how to allocate resources rationally, and how to avoid resource hogging. Moreover, resources should not be allocated to processes that are already in a waiting state. Thus, resource allocation must follow reasonable policies.

"Necessary conditions" is not a valid argument

- For formal systems/models:
  - We can directly prove whether the system is *deadlock-free*.
- For real-world complex systems:
  - Which condition is the easiest to break?

How to Avoid Deadlock in Real Systems?

Lock Ordering

- At any time, the number of locks in the system is finite
- Assign a unique ID to each lock (*Lock Ordering*)
  - Always acquire locks in increasing order of their IDs

Proof (Sketch)

- At any time, there is always a thread trying to acquire the highest-numbered lock
- That thread can always proceed

Data Race

(So if you don’t lock it, there will be no deadlock, right?)

Data Races

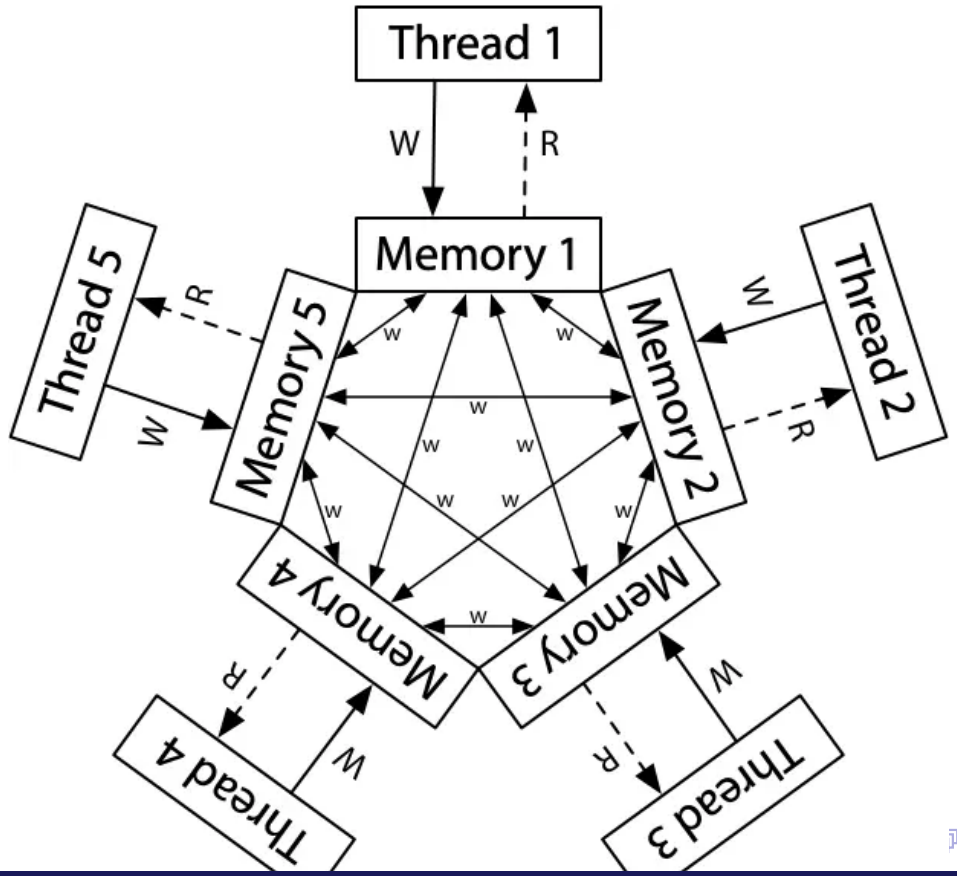
Different threads access the same memory location at the same time, and at least one access is a write.

- Two memory accesses ”race” — whichever happens first ”wins”
- Example: Peterson’s Algorithm implemented with shared memory

Data Races (cont'd)

"Winning the race" isn't as simple as it seems

- **Weak memory model** allows different observers to see different outcomes
- Since C11:
  If a data race occurs, the behavior of the program is undefined.



Data Races: Examples

The following code snippets cover the majority of data race scenarios you'll encounter.

- Don't laugh — most of your bugs are just variations of these two.

Case 1: Locked the wrong thing

```
void T_1() { spin_lock(&A); sum++; spin_unlock(&A); }
void T_2() { spin_lock(&B); sum++; spin_unlock(&B); }
```

Case 2: Forgot to lock

```
void T_1() { spin_lock(&A); sum++; spin_unlock(&A); }
void T_2() { sum++; }
```

An Example

Different threads access the same memory at the same time, and at least one access is a write.

```
void wait_next_beat(int expect) {
    // This is a spin-wait loop.
retry:
    mutex_lock(&lk);
    // This read is protected by a mutex.
    int got = n;
    mutex_unlock(&lk);

    // Case 2: forgot to lock
    if (n != expect) goto retry;
}
```

- Actually not a forgotten lock — the wrong variable was used
- More dangerous: this kind of bug (error state) is hard to trigger — a classic Heisenbug

Why Not Laugh?

Real systems face much more complex scenarios

- **"Memory"** could be any memory location in the address space:
  - Global variables
  - Heap-allocated variables
  - Stack variables
- **"Access"** could be any kind of code:
  - Could occur in your own source code
  - Could occur in framework/library code
  - Could be a line of assembly you never read
  - Could be a simple `ret` instruction

Key Insights

A significant number of concurrency bugs ultimately manifest as data races.

Protect shared data with locks / Eliminate all data races

For beginners in concurrent programming, it's important not only to intentionally avoid data races, but also to remember that forgetting to lock, using the wrong lock, or accessing shared resources outside of critical sections can all lead to data races

Atomicity/Ordering Violations

The Fundamental Difficulty

Humans are sequential creatures

- We can only understand concurrency in a sequential way
  - Programs are seen as composed of "blocks," and we assume each block runs uninterrupted (atomicity)
  - Example: `produce` $\rightarrow$ (happens-before) $\rightarrow$ `consume`

Concurrency control mechanisms are completely "self-managed", especially for C lanuage

- Mutual exclusion (lock/unlock) relies on the programmer:
  - Forgetting to lock $\rightarrow$ atomicity violation (AV)
- Condition variables/signaling (wait/signal) rely on the programmer:
  - Forgetting to signal $\rightarrow$ order violation (OV)

Threads cannot be implemented as a library

So... Are Programmers Really Doing It Right?

"Empirical Study" – Real-World Research

- Collected 105 concurrency bugs from real systems:
  - MySQL (14/9), Apache (13/4), Mozilla (41/16), OpenOffice (6/2)
- The study aimed to determine whether meaningful patterns exist

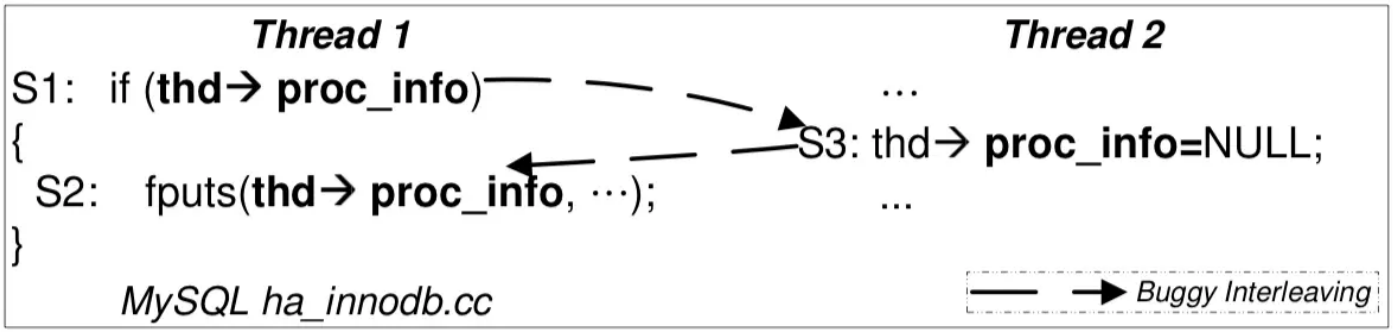
97% of non-deadlock concurrency bugs are atomicity or order violations

- Because "humans are sequential creatures"
- Learning from Mistakes – A Comprehensive Study on Real-World Concurrency Bug Characteristics
  - ASPLOS'08, Most Influential Paper Award

Atomicity Violation

"ABA": Code gets forcibly interleaved by another thread

- Even if locks are used (eliminating data races), AVs can still occur
- Examples:
  - Item duplication in *Diablo I*
  - Therac-25: "Move Mirror + Set State" sequence



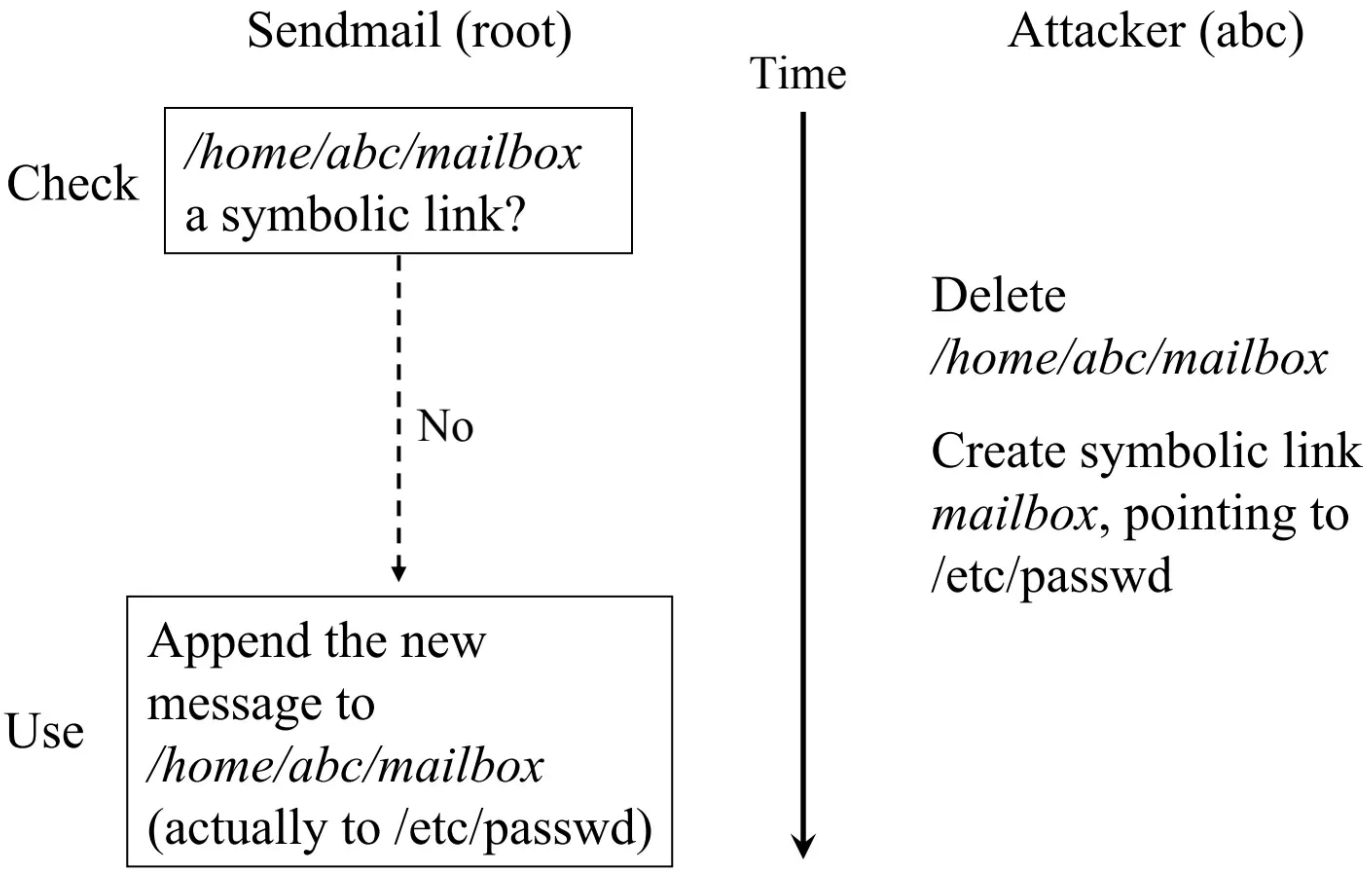
Example from: `MySQL ha_innodb.cc`

Event-Level Concurrency

```
Event(doMouseMove) {
    hoveredItem = Item("$1");
}

// Unexpected interleaved event
Event(clickEvent) {
    hoveredItem = Item("$99"); // <- Shared state
    Inventory.append(hoveredItem);
}

Event(doPickUp) {
    InHand = hoveredItem;
}
```

Atomicity Violation (cont'd)

Operating systems have even more shared state

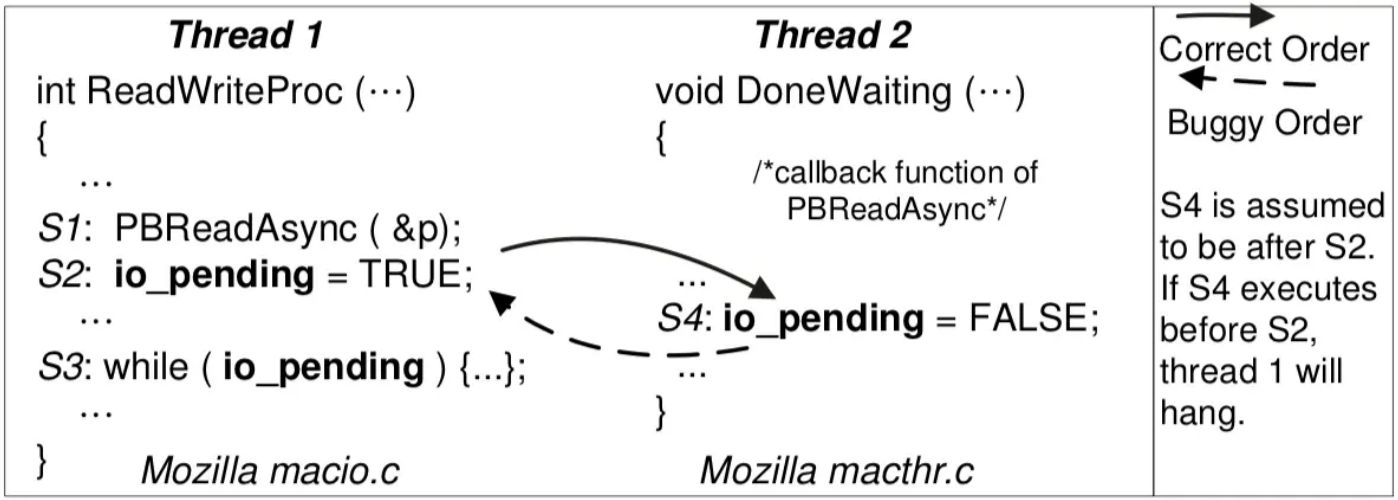


- TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study (FAST'05) — We can model this behavior

Order Violation

"BA": Events occur out of the expected order

- Example: *concurrent use-after-free*
- GhostRace (USENIX Sec'24)



Example from: Mozilla `macio.c` and `machtr.c`

Tips

- Atomicity has long been the ultimate goal of concurrency control. Ideally, a block of code should appear to either execute entirely in an instant or not at all. However, side effects — such as writes to shared memory or the file system — make atomicity difficult to achieve.
- Due to its appeal, atomicity has been a persistent pursuit at both the hardware and system levels: from database transactions (TX), to software- and hardware-supported transactional memory (an idea ahead of its time), to operating system transactions. Yet even today, we lack universally accessible and reliable atomicity guarantees for programmers.
- Guaranteeing execution order is even more challenging. Features like managed runtimes with automatic memory management and communication primitives such as channels aim to reduce programmer error in concurrent environments.

Takeaways

Human beings are fundamentally sequential creatures, and therefore tend to understand concurrent programs using a simplified model of “sequential blocks of execution.”
This mental model leads to two common types of concurrency bugs:

- Atomicity Violation: Code that is supposed to execute atomically gets interrupted.
- Order Violation: Code that is supposed to execute in a specific order fails to do so due to missing or incorrect synchronization.

A critical concept related to both of these bug types is the data race — when two threads access the same memory location simultaneously, and at least one of the accesses is a write. Data races are extremely dangerous, and we must strive to avoid them in our programs.