Lecture 19: Concurrency Bugs and Debugging

(Deadlocks, Data Races, Atomicity/Ordering Violations)

Xin Liu

Florida State University
xliu15@fsu.edu

COP 4610 Operating Systems
https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

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.
- At the same time, eliminating concurrency bugs during programming remains a global challenge. So, how should we deal with these concurrency issues?

Main Topics for Today:

- Deadlocks and How to Avoid Them
- Data Races
- Atomicity/Ordering Violations

Concurrency Bugs that Cost Lives

The First Computer "Bug" (1947)

- Date: September 9, 1947
- Machine: Harvard Mark II relay computer
- Team: Grace Hopper's group recorded a failure in a relay
- A moth was found in Relay #70, Panel F, and taped into the logbook
- Log note: *"First actual case of bug being found."*
- The word "bug" existed before 1947, but this event popularized it in computing
- Today, "bug" means a defect that can cause incorrect behavior or security risks

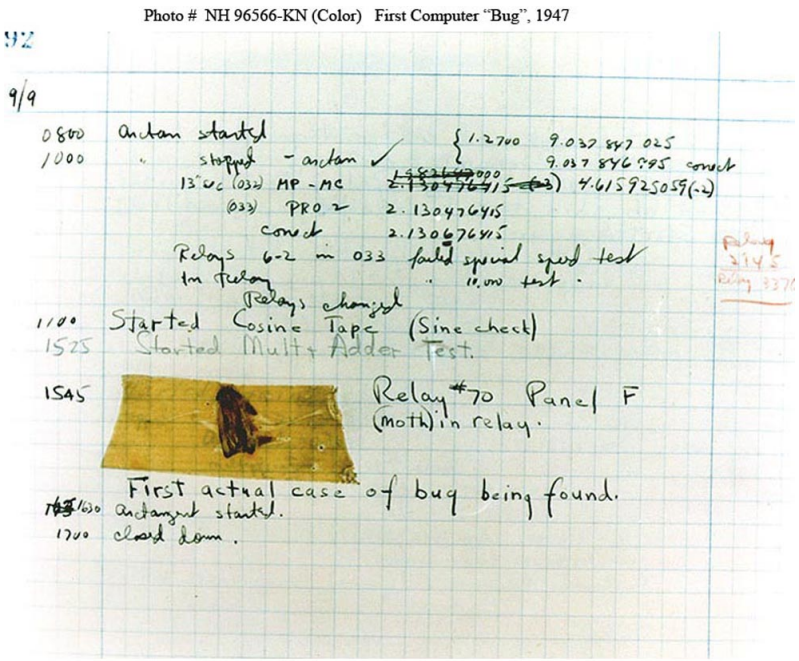


Figure: Harvard Mark II maintenance log page

A Retrospective: Item Duplication in Diablo I

How to dupe items in Diablo I

1. Drop the item you want to duplicate on the ground, walk a few steps away.
2. Click to **pick it up** (start walking toward it).
3. **At the exact moment** the pickup happens, click a belt potion slot.
4. The game mis-binds the "just-picked-up" item ID to the potion slot, so the potion **turns into a copy** of that item.
5. Drop the "potion" on the ground — it appears as the duplicated item; pick both up and repeat.

Intuition: a timing race between the ground-pickup handler and the belt-click handler causes the belt slot to adopt the item being picked up (shared cursor/item ID updated in the same tick).

Diablo I Bug: 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;
}
```

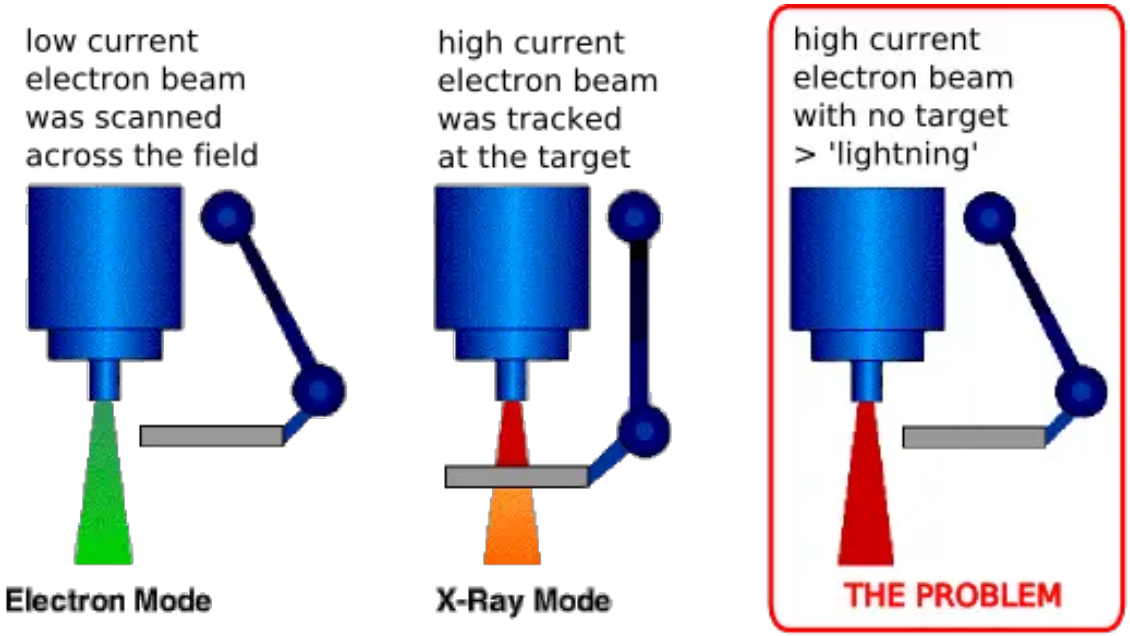
- Shared variable `hoveredItem` is overwritten by a late click.
- Pickup then reads the new value, not the hovered one.
- Inventory and hand both get `"$99"`.

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 and How to Avoid Them

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
```

In the interrupt handler, locking the same lock returns False because it is already held, so the handler spins forever, never returns to the thread to unlock, and the system stalls.

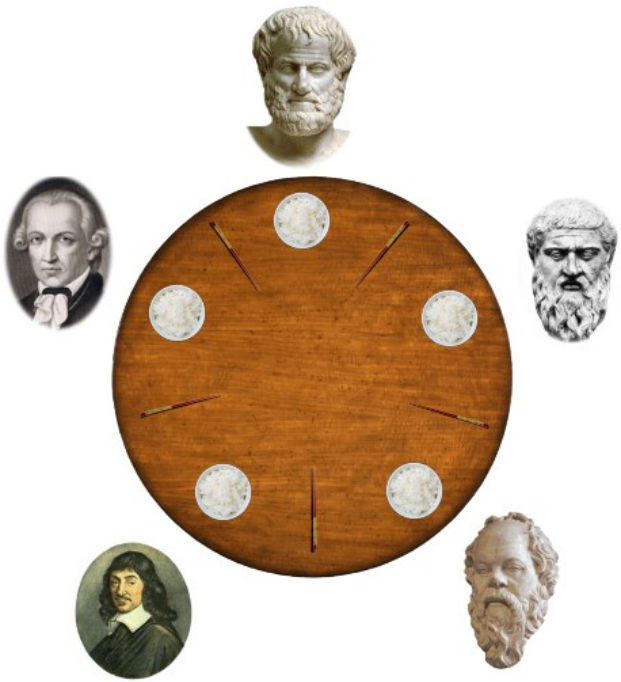
Looks silly? Think you won't make this mistake?

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

In-Class Quiz

Philosopher Dining Problem

- Philosophers (threads) alternate between thinking and eating.
- To eat, a philosopher must hold both the left and the right fork at the same time.
- If a fork is held by someone else, the philosopher must wait. How do we synchronize?
- How can we implement this using mutexes or semaphores?



A Failed Attempt

Have A Try: philosopher.c

```
void Tphilosopher(int id) {
    int lhs = (id + N - 1) % N;
    int rhs = id % N;
    while (1) {
        P(&avail[lhs]);
        printf("+ %d by T%d\n", lhs, id);
        P(&avail[rhs]);
        printf("+ %d by T%d\n", rhs, id);
        // Eat.
        // Philosophers are allowed to eat in parallel.
        printf("- %d by T%d\n", lhs, id);
        printf("- %d by T%d\n", rhs, id);
        V(&avail[lhs]);
        V(&avail[rhs]);
    }
}
```

When each philosopher grabs the left fork first, they all block waiting for the right fork, causing a deadlock.

A Working Pattern

If you want a working pattern, just use condition variables.

```
/* Try to take both forks */
pthread_mutex_lock(&mutex);

while (!(avail[lhs] && avail[rhs])) {
    pthread_cond_wait(&cv, &mutex);
}

avail[lhs] = avail[rhs] = false;    // reserve both

pthread_mutex_unlock(&mutex);

/* ... eat ... */

/* Release and notify */
pthread_mutex_lock(&mutex);
avail[lhs] = avail[rhs] = true;
pthread_cond_broadcast(&cv);        // wake all to recheck
pthread_mutex_unlock(&mutex);
```

No Semaphores: Let One Manager Handle the Forks

Leader / follower = producer / consumer

- A common pattern in distributed systems (e.g., HDFS): one coordinator grants access.
- Philosophers send requests; the waiter decides who may eat.

```
void Tphilosopher(int id) {
  send_request(id, EAT);
  P(allowed[id]); // The waiter hands the fork to the
    philosopher.
  philosopher_eat();
  send_request(id, DONE);
}

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

Forget the Fancy Synchronization Algorithms

You might worry that a single waiter (manager) is a performance bottleneck:

- A big table, everyone calling the waiter
- Premature optimization is the root of all evil (D. E. Knuth)

Start from the workload. Optimize only when needed.

- Eating time is usually much longer than the time to ask the waiter.
- If one manager cannot keep up, use more than one.
- Design the system so the coordinator does not become a bottleneck.
  - Millions of Tiny Databases (NSDI'20)

Necessary Conditions for Deadlock on Textbook

System Deadlocks (1971):

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

"Necessary" Conditions?

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

Easier said than done.

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.

However, "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?
    - Circular Chain!

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

Lock Ordering: Application

Linux Kernel: mm/rmap.c

```
/*
 * Lock ordering in mm:
 *
 * inode->i_rwsem       (while writing or truncating, not reading or faulting)
 *   mm->mmap_lock
 *     mapping->invalidate_lock (in filemap_fault)
 *       page->flags PG_locked (lock_page)
 *         hugetlbfs_i_mmap_rwsem_key (in huge_pmd_share, see hugetlbfs below)
 *           vma_start_write
 *             mapping->i_mmap_rwsem
 *               anon_vma->rwsem
 *                 mm->page_table_lock or pte_lock
 *                   swap_lock (in swap_duplicate, swap_info_get)
 *                     mmlist_lock (in mmput, drain_mmlist and others)
 *                     mapping->private_lock (in block_dirty_folio)
 *                       folio_lock_memcg move_lock (in block_dirty_folio)
 *                         i_pages lock (widely used)
 *                           lruvec->lru_lock (in folio_lruvec_lock_irq)
 *                     inode->i_lock (in set_page_dirty's __mark_inode_dirty)
 *                     bdi.wb->list_lock (in set_page_dirty's __mark_inode_dirty)
 *                       sb_lock (within inode_lock in fs/fs-writeback.c)
 *                       i_pages lock (widely used, in set_page_dirty,
 *                                 in arch-dependent flush_dcache_mmap_lock,
 *                                 within bdi.wb->list_lock in __sync_single_inode)
 *
 * anon_vma->rwsem,mapping->i_mmap_rwsem   (memory_failure, collect_procs_anon)
 *   ->tasklist_lock
 *     pte map lock
 *
 * hugetlbfs PageHuge() take locks in this order:
 *   hugetlb_fault_mutex (hugetlbfs specific page fault mutex)
 *     vma_lock (hugetlb specific lock for pmd_sharing)
 *       mapping->i_mmap_rwsem (also used for hugetlb pmd sharing)
 *         page->flags PG_locked (lock_page)
```

More On Locking Order in Practice

Unreliable Guide to Locking: Textbooks will tell you that if you always lock in the same order, you will never get this kind of deadlock. **Practice will tell you that this approach doesn't scale**: when I create a new lock, I don't understand enough of the kernel to figure out where in the 5000-lock hierarchy it will fit.

The best locks are encapsulated: they **never get exposed in headers**, and are **never held around calls to non-trivial functions outside the same file**. You can read through this code and see that it will never deadlock, because it never tries to grab another lock while it has that one. People using your code don't even need to know you are using a lock.

Deadlock: Dead Ends

Two sides: a complex system on one side, unreliable humans on the other.

- **Hope**
  - Mark “do one thing” and keep it uninterrupted.
- **Reality**
  - “Doing one thing” must be broken into many steps.
  - Each step must hold the correct locks, and as few locks as possible.

LockDoc (EuroSys’19)

- “Only 53 percent of the variables with a documented locking rule are actually consistently accessed with the required locks held.”

What We Can Do: LockDep

A simple idea

- Print a log on every `acquire`/`release`.
- If any thread shows both $A \rightarrow B$ and $B \rightarrow A$, report a deadlock.
  - This can raise false positives (e.g., when proper synchronization exists):$A \rightarrow B \rightarrow$ spawn $\rightarrow B \rightarrow A$

A more practical approach

- Maintain a dynamic *lock dependency graph* and do cycle detection.
- Have A Try: lockdep

Data Race

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

Data Races

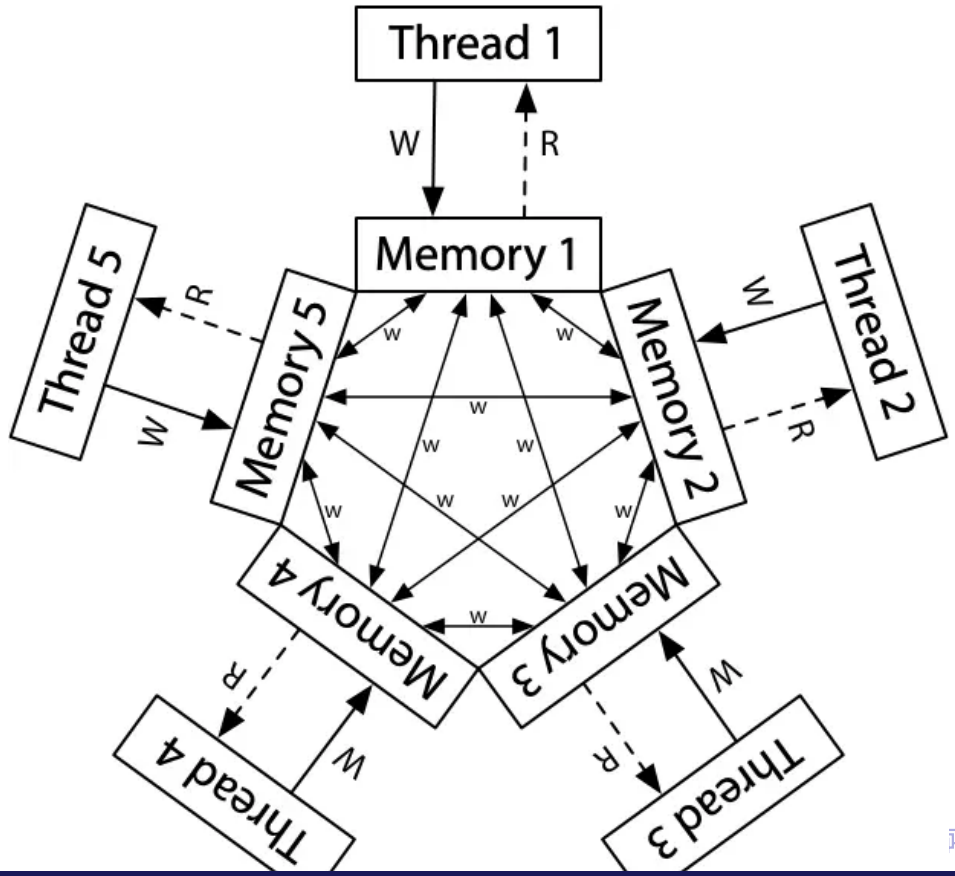
Different threads access the same memory location at the same time, and at least one access is a write.

- 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. It should be: `got != expect`
- 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

How to Avoid Data Races

Protect shared data with locks

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

Key Insights: A significant number of concurrency bugs ultimately manifest as 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

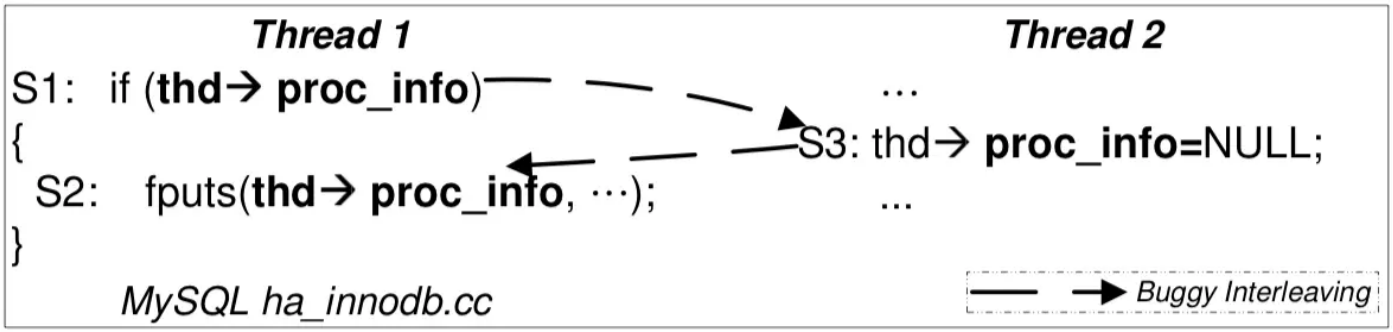
97% of non-deadlock concurrency bugs are atomicity or order violations

- 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

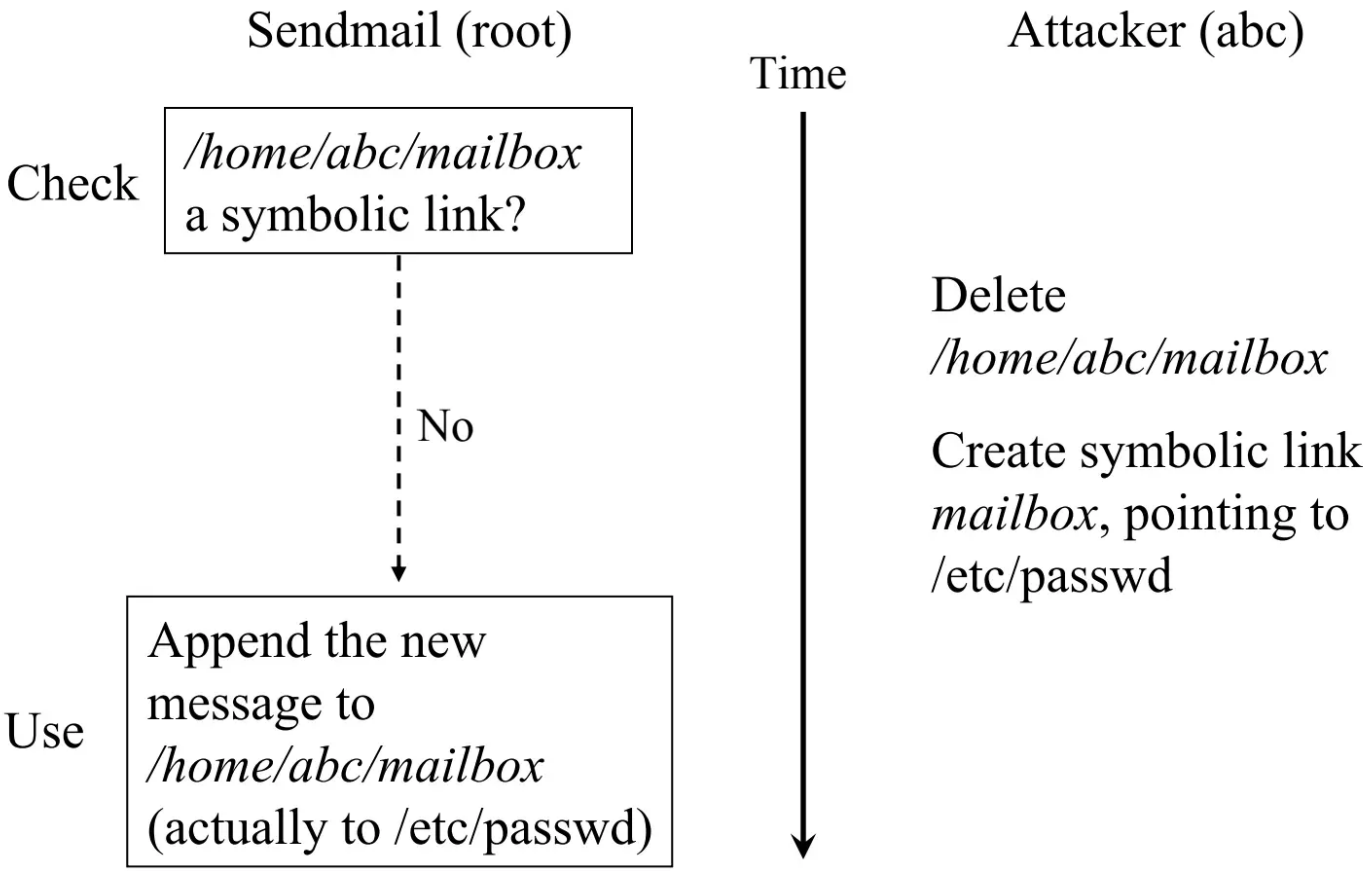
- Even if locks are used (eliminating data races), AVs can still occur
- Examples:
  - Item duplication in *Diablo I*
  - Therac-25: "Move Mirror + Set State" sequence



Example from: `MySQL ha_innodb.cc`

Atomicity Violation (cont'd)

Operating systems have even more shared state

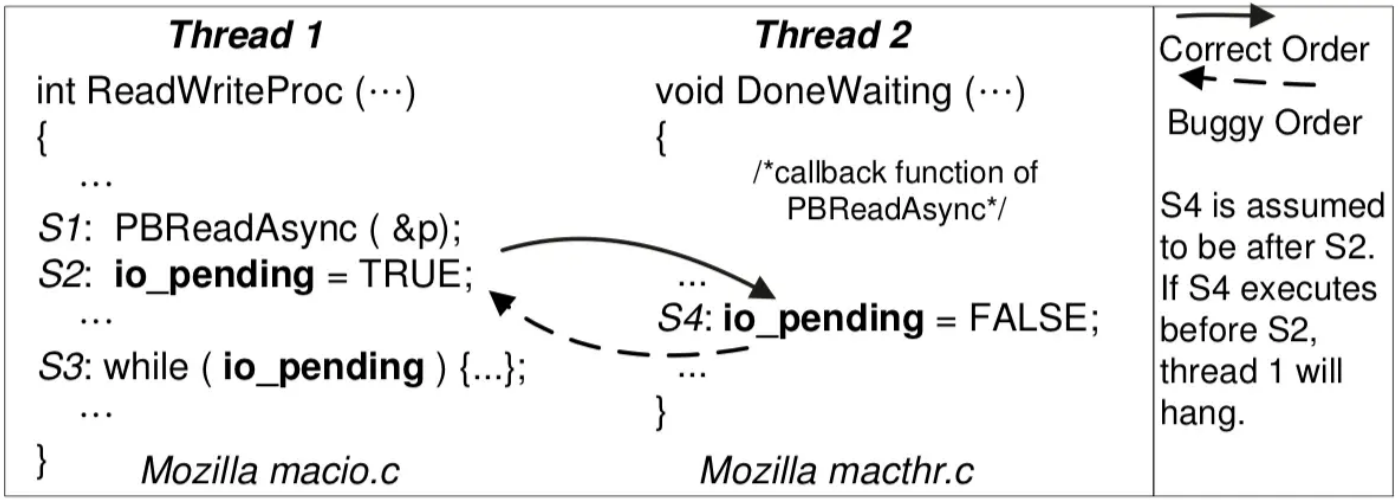


- TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study (FAST'05) — We can model this behavior

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.

Debugging Theory

Debugging Theory

Programs = A projection of the physical world into the digital world

- A **bug** = The breakdown of a programmer's assumptions about the "physical world"

From Requirements $\rightarrow$ Design $\rightarrow$ Code (Fault/Bug) $\rightarrow$ Execution (Error) $\rightarrow$ Failure

- We can only observe **failure** (visible incorrect outcomes)
- We can verify **correctness** of states — but it's expensive
- We often cannot locate the bug — every step may "look fine"

Concurrency bugs

- "Correctness" is not only hard to define — it's also hard to check

Inference

Program = A lossy projection of physical-world processes into the digital world

- As long as we "translate" into code — it may not match the actual requirements or constraints of the real world

Example: The balance field in "Fake eBay"

- Yes — this was a **data race**!
- 0 $\rightarrow$ `18446744073709551516`
- No one "normal" would consider this correct:
  - First, this is an **underflow**
  - A field like `balance` typically implies *no-underflow* by design
  - Also, this value increasing by a huge amount makes no sense

The Real "Software Crisis"

Specification Crisis

- Software specifications are inherently tied to the human world
- We cannot express all specifications in programming languages
- Even when we can write down a specification, it is often hard to prove

Search-space Curse

- The combinations of behaviors grow exponentially
- Even if you know the specification, it's extremely hard to verify that a complex system satisfies it
- Intuition: *"Complex systems are chaotic."*

Automatic Runtime Checking

The Programmer's Self-Rescue

We can check all well-defined specifications at runtime!

- Single/Mult- Thread style deadlocks
- Data races
- Signed integer overflows (undefined behavior)
- Use after free
- ...

Dynamic Program Analysis: A function $f(\tau)$ over the execution history of a state machine

- Pay the price of slowing down execution
- Find many more bugs

Runtime Lock Ordering Check

An Idea:

- For each acquire/release, record `tid` and `lock name`
- Assert: $G(V, E)$ has no cycles
  - V: all lock names
  - E: for every observation of holding u then acquiring v, add edge (u, v) to E

```
T1 ACQ a
T1 ACQ b
T1 REL b
T1 REL a
T2 ACQ b
T2 ACQ a
...
```

Lock analysis is essentially a dynamic graph problem.

Linux Kernel Lockdep

Solves the "naming" problem for locks

- The name can be the lock's address
- Or the site where the lock is initialized (stricter; fewer false positives)
  - The kernel lock validator
  - Since Linux Kernel 2.6.17, a big kernel lock $\rightarrow$ small kernel locks

Question

How do you name your locks **efficiently**?

ThreadSanitizer: Runtime Data Race Detection

Basic Idea

- A data race occurs when two threads access the same variable concurrently, and at least one access is a write
- Example: `T1: load(x);` `T1: t = t + 1;` `T2: store(x);` $\rightarrow$ still a data race

For any two accesses x, y on different threads (at least one write), check:

- A "happens-before" race
- Implemented using Lamport's Vector Clock: Time, clocks, and the ordering of events in a distributed system

Coping with Dead Ends: Sanitizers

Essential tools for modern, complex software systems

- AddressSanitizer (asan). Have A Try: asan
  - (paper): Detects illegal memory accesses
  - Handles: heap/stack/global buffer overflows, use-after-free, use-after-return, double-free, ...
  - Note: No KASAN → Linux kernel's quality/security becomes fragile
- ThreadSanitizer (tsan). Have A Try: tsan
  - Detects data races
  - KCSAN: Concurrency bugs should fear the big bad data-race detector
- MemorySanitizer (msan), UBSanitizer (ubsan), ...
- **SpecSanitizer**: Based on AI/LLM-style "specification checking"
  - Stay tuned...

When there is a specification, developers will naturally think of appropriate methods to verify it, in order to improve software quality.
The C language only provides low-level mechanisms, which become inadequate when managing large and complex projects.
Without mechanisms like ASAN and TSAN, the Linux kernel would be riddled with bugs.

Defensive Programming

Believe in Your Programming Skills!

Full Sanitizer is hard to implement

- Maybe it's time to think differently
- After all, we can **code**!

Best-effort is better than no-effort!

- Not aiming for **complete** checks (some false positives/negatives are fine)
- But even simple checks can be **very effective** — and surprising!
  - Haven't we always written assertions?
  - Peterson's algorithm: `assert(next == 1);`
  - Linked list: `assert(u->prev->next == u);`
  - spinlock: `if (holding(&lk)) panic();`

Defensive Programming: Buffer Overflow Detection

Canary — birds highly sensitive to carbon monoxide

- They gave their lives to warn miners of toxic gas leaks underground (since 1911)

Canary: "sacrificial" memory cells that warn of memory errors

- (Like in real life: the program continues running until a sentinel value is corrupted)

Example of a Canary: Stack Guard

Stack overflow:

```
#define MAGIC 0x55555555
#define BOTTOM (STK_SZ / sizeof(u32) - 1)
struct stack { char data[STK_SZ]; };

void canary_init(struct stack *s) {
    u32 *ptr = (u32 *)s;
    for (int i = 0; i < CANARY_SZ; i++)
        ptr[BOTTOM - i] = ptr[i] = MAGIC;
}

void canary_check(struct stack *s) {
    u32 *ptr = (u32 *)s;
    for (int i = 0; i < CANARY_SZ; i++) {
        panic_on(ptr[BOTTOM - i] != MAGIC, "underflow");
        panic_on(ptr[i] != MAGIC, "overflow");
    }
}
```

Another Kind of Canary

Detecting "Buffer Overflow"

```
int foo() {
    // Canary placed in local memory: between local vars
   and return address
    u32 canary = SOME_VALUE;

    ... // actual function logic

    // if overwritten by attack, result != 0
    // intact canary will XOR back to 0
    canary ^= SOME_VALUE;

    assert(canary == 0);
    return ret;
}
```

0xCC, 0xDD, and 0xFD: The Unsung Heroes

Guard/fence/canary patterns in MSVC Debug Mode

- Uninitialized stack: `0xcccccccc`
- Uninitialized heap: `0xcdcdcdcd`
- Object head/fence: `0xfdfdfdfd`
- Freed memory: `0xdddddddd`
  - If you see these in the debugger: it's them protecting you!

Defensive Programming: Lightweight Lockdep

Is full-featured lockdep too complicated?

- Count the current spin attempt
- If it exceeds an obviously abnormal number (e.g., 100,000,000), report an error
  - That's when you start to feel "hangy"

```
int spin_cnt = 0;
while (xchg(&lk, X) == X) {
    if (spin_cnt++ > SPIN_LIMIT) {
        panic("Spin limit exceeded @ %s:%d\n",
              __FILE__, __LINE__);
    }
}
```

- Combine with debugger + thread backtrace to diagnose deadlocks in one second

L1 Memory Allocator Specification

- Allocated memory set: $S = [\ell_0, r_0) \cup [\ell_1, r_1) \cup \ldots$
- The range `kalloc(s)` returns, $[\ell, r)$, must satisfy:

$$[\ell, r) \cap S = \emptyset$$

```
// allocation
for (int i = 0; (i + 1) * sizeof(u32) <= size; i++) {
    panic_on(((u32 *)ptr)[i] == MAGIC, "double-allocation
    ");
    arr[i] = MAGIC;
}

// free
for (int i = 0; (i + 1) * sizeof(u32) <= alloc_size(ptr);
    i++) {
    panic_on(((u32 *)ptr)[i] == 0, "double-free");
    arr[i] = 0;
}
```

Lightweight ThreadSanitizer

Recap: How data races manifest

- The result of a race can affect observable program state
- So if we can observe state differences, we can catch the race!

```
// Suppose x is lock-protected
...
int observe1 = x;
delay();
int observe2 = x;

assert(observe1 == observe2);
...
```

Effective data-race detection for the Kernel (OSDI'10)

Defensive Programming: SemanticSanitizer

Two simple-looking checks:

- Check if an integer is within a valid range

```
#define CHECK_INT(x, cond) \
  ({ panic_on(!(x cond), \
    "int check fail: " #x " " #cond); })
```

- Check whether a pointer is in the heap

```
#define CHECK_HEAP(ptr) \
  ({ panic_on(!IN_RANGE((ptr), heap)); })
```

How should this be used?

Defensive Programming: SemanticSanitizer (cont'd)

Check internal data consistency

- $CHECK_INT(waitlist->count, >= 0);$
- $CHECK_INT(pid, < MAX_PROCS);$
- $CHECK_HEAP(ctx->rip); CHECK_HEAP(ctx->cr3);$

Act as a "variable contract"

- $CHECK_INT(count, >= 0);$
- $CHECK_INT(count, < 10000);$

Helps prevent many types of bugs — even partially covers what **AddressSanitizer** can do:

- Overflow, use-after-free — all may lead to memory corruption

This Is Real "Programming"

With just a few lines of code, we've implemented:

- Stack guard
- Lockdep (simple)
- AddressSanitizer (simple)
- ThreadSanitizer (simple)
- SemanticSanitizer

These are seeds:

- They point toward the limitless space of real *engineering*
- And they invite us to rethink the design of programming languages

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.

Takeaways

Bugs — including concurrency bugs — have long plagued practitioners across all areas of software engineering.
We not only face the specification crisis — the challenge of defining what is ”correct” — but even when a specification is known, the complexity of modern software systems can still overwhelm us.
To cope, we’ve developed a practical compromise: instead of letting programs silently drift into failure, we encode expectations directly into the program itself — through invariants like race-freedom, lock ordering, and more.
The lesson from our ”bootleg” sanitizers is this: If we can clearly trace the root cause of a problem, we can always find a good solution. These lightweight tools quietly help you build fail-fast systems, reducing the burden of debugging and making bugs surface early.
I hope this lecture inspires and helps you rethink what it truly means to ”program.”