Lecture 19: Concurrency Bugs and Debugging

(Deadlock and Defensive Programming)

Xin Liu

Florida State University
xliu15@fsu.edu

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

Outline

Review:

- Fundamental tools for concurrent programming
  - Thread libraries
  - Mutual Exclusion (e.g., spinlocks, mutexes)
  - Problem modeling methods (e.g., producer-consumer problem)
  - Synchronization (e.g., condition variables, semaphores)
- Application scenarios for concurrent programming:
  - High-Performance Computing
  - Data Centers
  - Web/Mobile Applications

Today's Key Question:

Concurrent programming is so difficult,
what should I do when I encounter bugs?

Main Topics for Today:

- Deadlocks and data races
- Methods for dealing with bugs (including concurrency bugs)

Methods for Dealing with Bugs

Basic Approach: Doubt Yourself

Although it's hard to admit, always assume your code is wrong.

Then what?

- Write good tests
- Check where things went wrong
- Check again where things went wrong
- Check once more where things went wrong
- **Check every situation you think is "not quite right."**

The Root Cause of Bugs

Software is a projection of requirements (specifications) into the digital world of computers.

What often happens:

- Developers only focus on ”translating” code without ensuring it matches the actual requirements (specifications).

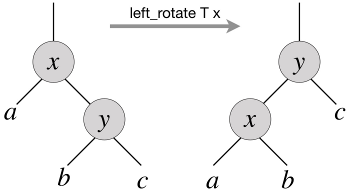
Example: producer-consumer problem

- In the real world, how could a non-existent item possibly be consumed?
- In the world of programs, an item is merely a symbol, losing the properties it has in the real world.

Express the conditions the program must satisfy using `assert`.

Examples:

- Rotation of a binary tree
- Use assertions to verify the correctness of node reordering during a rotation.



Specifications

You know the meaning of many variables:

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

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

- Variables have ”typed annotation.”

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

Why are these checks important?

- When the meaning of a variable changes, strange issues may arise (e.g., overflow, memory errors).
- Don’t underestimate these checks; they are common in low-level programming (M2, L1, ...).

Concurrency Bug: Deadlock

Recap: Modeling Concurrent Problems

Producer-Consumer Problem

- A fundamental synchronization problem that allows you to solve 99.9% of real-world concurrency issues.

Dining Philosophers Problem

- Another classic problem that demonstrates how multiple entities share limited resources (like CPUs).

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

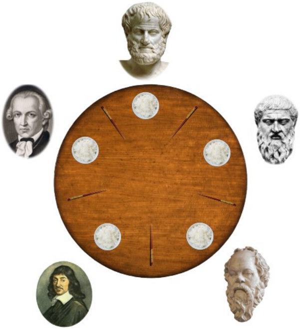


Illustration of the Dining Philosophers Problem

Failed Attempt:

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

```c
// 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

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

Concurrency Bug: Data Race

If we don't lock, we won't have deadlocks, right?

Data Race

A data race occurs when different threads access the same memory simultaneously, and at least one of them is writing.

What happens:

- Two memory accesses are ”racing” against each other, and the ”winner” executes first.

Differences between Data Race and Race Condition:

- **Race Condition** is a broader term that refers to any situation where the outcome depends on the timing or order of execution of threads.
- **Data Race** is a specific type of race condition, where multiple threads access the same memory location concurrently, with at least one thread writing and without proper synchronization.
- **Race Condition** may or may not involve shared data, while **Data Race** specifically involves concurrent read/write access to shared memory.
- Proper synchronization (e.g., using locks or atomic operations) can eliminate a data race but not necessarily all race conditions.

Peterson's Algorithm teaches us:

- Writing correct lock-free concurrent programs is incredibly hard.
- Ironically, this makes things simpler!

The Solution:

- Use mutexes to protect shared data.
- Eliminate all data races.

Mutual exclusion is the key to making concurrent programs reliable.

Data Race: Examples

These code snippets summarize most of the data race cases you'll encounter:

Don't laugh — your bugs are almost always variations of these two cases!

```
// Case #1: Wrong lock
void thread1() { spin_lock(&lk1); sum++; spin_unlock(&lk1); }
void thread2() { spin_lock(&lk2); sum++; spin_unlock(&lk2); }

// Case #2: Forgot to lock
void thread1() { spin_lock(&lk1); sum++; spin_unlock(&lk1); }
void thread2() { sum++; }
```

More Types of Concurrency Bugs

Programmers: Creative Ways to Make Mistakes

Review of the tools we use for concurrency control:

- **Mutexes** (lock/unlock) - Ensure atomicity
- **Condition variables** (wait/signal) - Ensure synchronization

Common mistakes:

- Forgetting to lock — Atomicity Violation (AV)
- Forgetting to synchronize — Order Violation (OV)

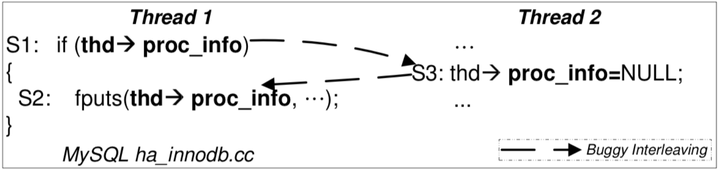
Empirical study:

- In 105 concurrency bugs (non-deadlock/deadlock):
  - MySQL: 14 AV / 9 deadlock
  - Apache: 13 AV / 4 deadlock
  - Mozilla: 41 AV / 16 deadlock
  - OpenOffice: 6 AV / 2 deadlock
- 97% of non-deadlock concurrency bugs are either AV or OV.

"ABA" Problem:

- You thought your piece of code was safe, but it was interrupted by another thread.
- The sequence appears to be correct (A $\rightarrow$ B $\rightarrow$ A), but the intermediate state leads to unexpected behavior.

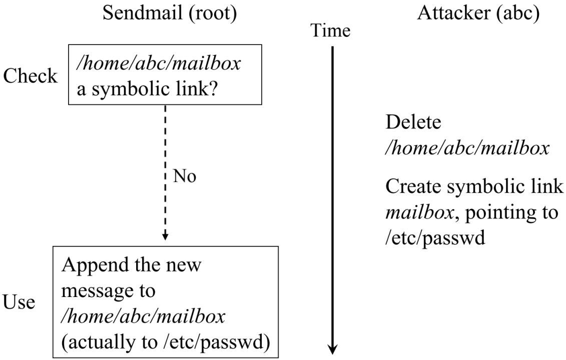
I thought everything was fine, but someone aggressively intervened.



Sometimes locking doesn't solve the problem:

"TOCTTOU" - Time of Check to Time of Use:

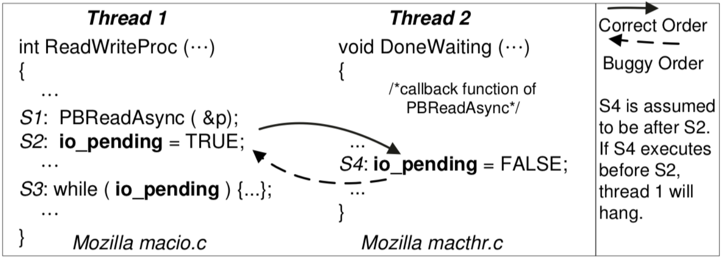
- Even if you use locks, there can be a gap between checking a condition and using the result.
- This can lead to atomicity violations if the state changes during this gap.



"BA" Problem:

- Why didn't things happen in the order I expected?
- This happens when the execution order of operations is reversed from what you intended.

Example: Concurrent use after free



Lockdep: Runtime Deadlock Detection

Lockdep Specification:

- Assign a unique ”allocation site” for each lock.
- **Assertion:** There must be a globally unique locking order for locks from the same allocation site.

How to check:

- Use `printf` to record all observed locking orders.
- Example:
  - Observed order: `[x, y, z]`
  - Implies the following locking relationships:
    - $x \rightarrow y$
    - $x \rightarrow z$
    - $y \rightarrow z$

ThreadSanitizer: Runtime Data Race Detection

What is ThreadSanitizer?

- A tool for detecting **data races** and **concurrency issues** in multi-threaded programs.
- Supported in **C**, **C++**, and **Go** programming languages.

How Does ThreadSanitizer Work?

- Uses dynamic analysis to monitor **memory access** by different threads.
- Establishes a **happens-before** relationship using program-order and synchronization operations.
- Detects conflicting access (read/write) that are not properly synchronized, which results in **data races**.

Output:

- Provides detailed reports showing where data races or other concurrency issues occurred.
- Helps developers **identify and fix** potential issues in multithreaded applications.

Canary

Canary: Sensitive to carbon monoxide

- Historically, canaries were used to warn miners of gas leaks by being more sensitive to toxic gases (since 1911).

Canary in Computer Systems:

- In computer systems, a ”canary” sacrifices some memory cells to detect memory errors early.
- The canary value is placed in memory; if it changes unexpectedly, it indicates a buffer overrun.
- **(No animals were harmed during the program’s execution!)**

Canary Example: Protecting Stack Space (M2/L2)

```
#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");
  }
}
```

Explanation:

- The `MAGIC` value is placed at both ends of the stack to detect

Takeaways

Q: How can we save humanity from its weaknesses in concurrent programming?

Take-away message:

- **Common concurrency bugs:**
  - Deadlocks
  - Data races
  - Atomicity/Order violations
- Don't blindly trust yourself: **Check, check, check!**
- **Defensive programming:** Always validate conditions and assumptions.
- **Dynamic analysis:** Use logging and checks to detect issues early.