Lecture 22: Dealing with Concurrency Bugs

Deadlocks, Data Ends, Deadlines

Xin Liu

Florida State University

xliu15@fsu.edu

CIS 5370 Computer Security

https://xinliulab.github.io/cis5370.html

Outline

We’ve already covered the common types of concurrency bugs: deadlocks, data races, and atomicity/order violations. At the same time, eliminating concurrency bugs during programming remains a global challenge. So, how should we deal with these concurrency issues?

Today’s Key Question: A truly practical programming lesson — How to write correct (concurrent) programs by tackling the three most common forms of “death” in systems:

Main Topics for Today:

- Deadlock – system stuck waiting on itself
- Dead end – the software crisis: no way forward
- Deadline – running out of time

Review: Debugging Theory

Review: 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 � Design � Code (Fault/Bug) � Execution (Error) � 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 “MountainPay”

- (Yes — this was a **data race**!)
- 0 � `18446744073709551616`
- 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.”*

Deadlock Avoidance

Deadlock: A ”Simple” Concurrency Bug

Clearly Defined Specification

- Any thread should make progress under a ”reasonable” level of concurrency

Clearly Defined Necessary Conditions

1. **Mutual Exclusion** – Only one thread can hold a lock at a time
2. **Wait-for** – A thread holding a lock may wait for more
3. **No Preemption** – A lock cannot be forcibly taken away
4. **Circular Chain** – A circular waiting relationship is formed

Lock Ordering: Avoid Circular Wait

- Always acquire locks in a globally consistent order based on lock numbering

Lock Ordering: Application

Linux Kernel: `mm/rmap.c`

```
// Lock ordering in mm:
//
// inode->i_rwsem (while writing or truncating, not reading or faulting)
// -> mmap_lock
// -> i_mmap_rwsem
// -> page->flags PG_locked (lock_page)
// -> hugepage lock (in huge_pmd_share, see hugeTLBs below)
// -> anon_vma->rwsem
// -> mapping->invalidate_lock
// ...
// HugeTLBs (hugepage) take locks in this order:
// hugetlb_lock (hugetlbfs specific page fault mutex)
// vma_lock (mmap_lock for rmap sharing)
// page->flags PG_locked (lock_page)
```

However...

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: A Dead End

One side is a complex system, the other is unreliable humans.

- **The hope:**
  - Mark a ”do-one-thing” block and expect it not to be interrupted
- **The reality:**
  - ”Doing one thing” must be decomposed into many steps
  - Each step must be protected with the **correct** (and minimal) locks

LockDoc (EuroSys’19)

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

Automatic Runtime Checking

Self-Rescue

We can check all well-defined specifications at runtime!

- AA/ABBA-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
...
```

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, also in OpenHarmony v1

We also have a student-built implementation!

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);` � still a data race

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

$$x \not\preceq y \wedge y \not\preceq x$$

- This is called 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)
  - (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)
  - 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...

Defensive Programming in Operating Systems Labs

The Dilemmas Faced by Students

Great in theory, harsh in reality

- Our framework runs directly on virtual machines:
  - It doesn't support tools like **sanitizers**
  - Porting them over = failure
- It's practically impossible for us to "monitor every shared memory access"

***Operating Systems* Lab Deadlines**

- Students crash straight into them
- Deadlines are the number one source of panic

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 Overrun 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: If you’ve hit this... you probably spent half a day debugging it

```
#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 “epilogue overflow”

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

   ... // actual function logic

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

   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!
  - Devs joke about the “electronic chants” `cc dd fd dd dd...`
  - They’ve always been quietly guarding you from memory bugs.

```
for i in [0xcc, 0xcd, 0xdd, 0xfd]:
   print((bytes([i]) * 80).decode(’gbk’))
```

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

AddressSanitizer

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;
}
```

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

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.”