

# 并发bug

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组



# 并发bug

- 学习了并发之后，就知道编正确的并发程序实在是很困难
- 理解最简单的程序都很费劲
  - `pid = fork(); if (pid == 0) fork(); printf("哈哈哈哈哈不好笑\n");`
  - `P(empty); V(fill); P(fill); V(empty);`
  - 期中考试题各种千奇百怪的bug
  - 上百万、上千万行的代码怎么可能没有bug??
- 人类是聪明的
  - 既然知道错误无法避免，那当然是享受检测或者避免它们了
  - 我们就用编程语言本身做武器，和这些bug斗智斗勇.....

# 例子: RAI

- Resource Acquisition Is Initialization (RAII)
  - 一个常见的编程技巧 (曾经经常出现在大型项目的代码里, 现在已经是C++标准的一部分了)
  - 比起Linux Kernel里到处是goto要好不少

```
void write_to_file (const std::string &message) {  
    // mutex to protect file access (shared across threads)  
    static std::mutex mutex;  
    // lock mutex before accessing file  
    std::lock_guard<std::mutex> lock(mutex);  
    // try to open file  
    std::ofstream file("example.txt");  
    if (!file.is_open())  
        throw std::runtime_error("unable to open file")  
    // write message to file  
    file << message << std::endl;  
    // resource release regardless of exception  
}
```

# 死锁



# 我等你、你等我

```
void thread1() {  
    lock(&mutex1);  
    lock(&mutex2);  
    unlock(&mutex2);  
    unlock(&mutex1);  
}
```

```
void thread2() {  
    lock(&mutex2);  
    lock(&mutex1);  
    unlock(&mutex1);  
    unlock(&mutex2);  
}
```

- 更复杂的

- t1: lock(A) lock(B)
- t2: lock(B) lock(C)
- t3: lock(C) lock(A)

# 谁能保证把程序写对呢？

- Linux Kernel里有接近40,000个spin\_lock调用
  - 根本不是什么lock(A) lock(B)能描述得了的
  - 几乎每个内核对象都自带自旋锁
- 仅仅依靠程序员“自觉”避免问题根本就是天方夜谭
  - RAII也无法解决这个问题
- 一个自然的想法
  - 在lock(lk)的时候printf("lock lk");
  - 如果看到 t1: [A B] t2: [B A], 哪怕系统没有实际死锁, 坏事也没准要发生

# It Works!

- 是的，这么个工具能帮我们检测尚未发生的死锁
- 很好，但效率不太够
  - 思考题：怎么实现更高效的死锁检测(预测)?

# Linux Kernel lockdep

- 修改printf的粒度
  - lock(spinlock\_t \*lk) – printf(lk) 系统中可能有很多锁
  - 但我们不妨让所有类型的锁都看成同一个锁
  - 在lock分配的时候，给它一个唯一的key

```
#define spin_lock_init(lock) \  
do { \  
    static struct lockdep_type_key __key; \  
    __spin_lock_init((lock), #lock, &__key); \  
} while (0)
```

- 在运行时检查是否存在[A B] [B C] [C A]的情形



## Linux Kernel lockdep (cont'd)

- 图中的环路检测  $O(m) = O(n^2)$  -- 锁可能非常多
  - 如果每次lock/unlock都检查，代价太大
  - 如果总是记下来，日志也没地方放
- 每个线程维护一个lock stack
  - [A B] [B C] [C A]
  - 维护这个stack的hash，这样[A B]发生的时候更新一次，以后[A B]再发生就不更新了
- 就是这样——听起来高大上，一点也不神奇
  - 检测Linux Kernel内核死锁：问题求解内容 + 工程实践

死锁以外：原子性/顺序

# 回顾一下我们实现并发控制的工具

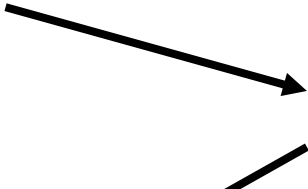
- lock/unlock
  - 实现原子性
- wait/notify/P/V
  - 实现同步(约束执行顺序)
- 如你所见，并发程序中最常见的两种bug
  - 原子性违反(Atomicity Violation, AV)
  - 顺序违反(Order Violation, OV)

Shan Lu, et al. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of ASPLOS*, 2008.

# Atomicity Violation: 例子

- 看起来非常蠢
  - 但你怎么知道几十万行代码里，它就没有呢？
  - MySQL ha\_innodb.cc

```
if (thd->proc_info) {  
    thd->proc_info = NULL;  
    fputs(thd->proc_info, ...);  
}
```



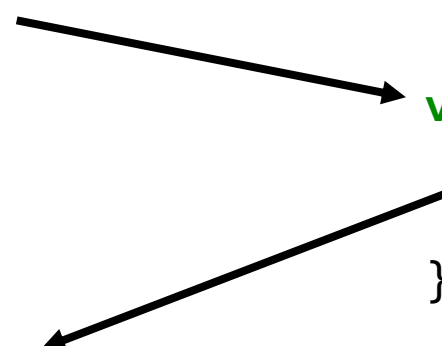
其他类似问题

```
sum++;  
buf[pos++] = data;
```

# Order Violation: 例子

- 设置一个flag, 然后等它好
  - 现实: flag丢了
  - Mozilla macio.c/macthr.c

```
int ReadWriteProc(...) {  
    PBReadAsync(&p);  
  
    io_pending = true;  
    while (io_pending) {  
        ...  
    }  
}  
  
void DoneWaiting(...) {  
    io_pending = false;  
}
```



# 有趣的Findings

- 基于报告的bugs
  - (Non-deadlock/deadlock): MySQL (14/9), Apache (13/4), Mozilla (41/16), OpenOffice (6/2)
- 有趣的发现：能帮我们找到更多bug吗？
  - Almost all (97%) of the examined non-deadlock bugs belong to one of the two simple bug patterns: atomicity-violation or order-violation.
  - Almost all (96%) of the examined concurrency bugs are guaranteed to manifest if certain partial order between 2 threads is enforced.
  - Many (66%) of the examined non-deadlock concurrency bugs' manifestation involves concurrent accesses to only one variable.
  - Almost all (97%) of the examined deadlock bugs involve two threads circularly waiting for at most two resources.

## Findings: 幸存者偏差

- 论文的结论针对已经报告的bug成立
- 但还有另一种可能
  - 非AV/OV的bug也许有，但开发者还没发现
  - 需要 $> 2$ 个线程触发的bug也许有，但开发者还没发现
    - 我们设计了个算法能发现！

# 数据竞争





## 另一种并发Bug的常见形态

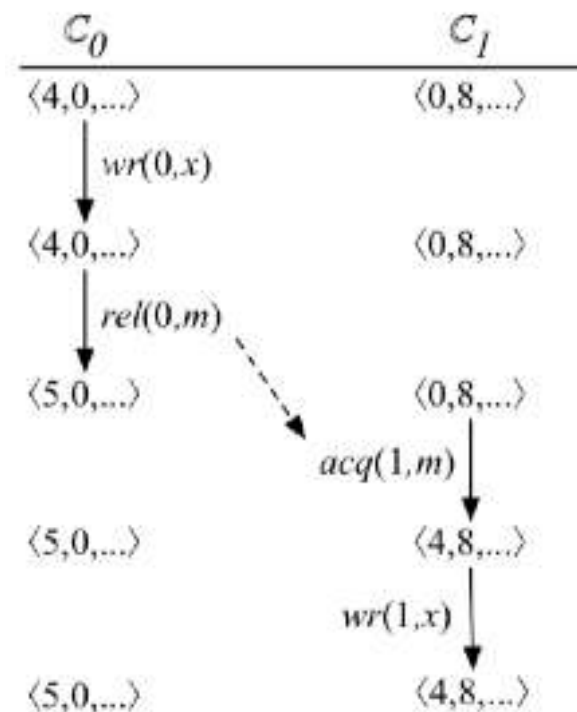
- 竞争条件(Race Conditions)/数据竞争(Data Race)
  - 两个内存访问
  - 同时发生在两个不同线程
  - 至少有一个是写
- AV/OV很大程度上是data race (但也有不是的情况)

# 数据竞争：检测

- 在库函数层提供内建机制，当发现bug苗头时立即报告
- 例子：Java ConcurrentModificationException
  - Java标准库的Collections不是线程安全的(LinkedList, ArrayList, TreeSet, HashMap, ...)
  - 一个线程在迭代时，另一个线程如果修改，就会抛出异常(不保证抛出)
  - 警告：你在搬石头砸自己的脚！

# 数据竞争：检测

- 同步操作规定了一些顺序
  - 其他事件可以“乱序”
  - 因此我们可以预测出数据竞争的存在
- 这就成了一个算法问题
  - 至今也是研究的热点



Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of PLDI*, 2009.



# 小结

- 和bug斗智斗勇：代码 = 武器(这些技术都可以用在oslab里)
  - 软件/系统研究：Fun and Profits
- 各种各样有趣的工具
  - RAIL (管理局部资源分配/释放)
  - Linux Kernel lockdep (死锁检测)
  - Race early fail / HB race detector (检测数据竞争)
- 更多更有趣的工具
  - Record & Replay, Transactional Memory, ...