互斥问题

并发编程：从入门到放弃

人类是 sequential creature

- 编译优化 + weak memory model 导致难以理解的并发执行
- 有多难理解呢?
  - Verifying sequential consistency 是 NP-完全问题

人类是 (不轻言放弃的) sequential creature

- 有问题，就会试着去解决
- 手段：“回退到”顺序执行
  - 标记若干块代码，使得这些代码一定能按某个顺序执行
  - 例如，我们可以安全地在块里记录执行的顺序

回退到顺序执行：互斥

插入“神秘代码”，使得所有其他“神秘代码”都不能并发

- 由“神秘代码”领导不会并发的代码 (例如 pure functions) 执行

```
void Tsum() {
  stop_the_world();
  // 临界区 critical section
  sum++;
  resume_the_world();
}
```

Stop the world 真的是可能的

- Java 有“stop the world GC”
- 单个处理器可以关闭中断
- 多个处理器也可以发送核间中断

失败的尝试

```
int locked = UNLOCK;

void critical_section() {
retry:
  if (locked != UNLOCK) {
    goto retry;
  }
  locked = LOCK;

  // critical section

  locked = UNLOCK;
}
```

和“山寨支付宝”完全一样的错误

- 并发程序不能保证 load + store 的原子性

更严肃地尝试：确定假设、设计算法

假设：内存的读/写可以保证顺序、原子完成

- `val = atomic_load(ptr)`
  - 看一眼某个地方的字条 (只能看到瞬间的字)
  - 刚看完就可能被改掉
- `atomic_store(ptr, val)`
  - 对应往某个地方 “贴一张纸条” (必须闭眼盲贴)
  - 贴完一瞬间就可能被别人覆盖

对应于 model checker

- 每一行可以执行一次全局变量读或写
- 每个操作执行之后都发生 `sys_sched()`

正确性不明的奇怪尝试 (Peterson 算法)

A 和 B 争用厕所的包厢

- 想进入包厢之前，A/B 都首先举起自己的旗子
  - A 往厕所门上贴上 “B 正在使用” 的标签
  - B 往厕所门上贴上 “A 正在使用” 的标签
- 然后，如果对方举着旗，且门上的名字是对方，等待
  - 否则可以进入包厢
- 出包厢后，放下自己的旗子 (完全不管门上的标签)

习题：证明 Peterson 算法正确，或给出反例

进入临界区的情况

- 如果只有一个人举旗，他就可以直接进入
- 如果两个人同时举旗，由厕所门上的标签决定谁进
  - 手快 有 (被另一个人的标签覆盖)、手慢 無

一些具体的细节情况

- A 看到 B 没有举旗
  - B 一定不在临界区
  - 或者 B 想进但还没来得及把 “A 正在使用” 贴在门上
- A 看到 B 举旗子
  - A 一定已经把旗子举起来了
  - (!@^#&!%^(&^!@%#

绕来绕去很容易有错漏的情况

Prove by brute-force

- 枚举状态机的全部状态 $(PC_1, PC_2, x, y, turn)$
- 但手写还是很容易错啊——可执行的状态机模型有用了!

```
void TA() { while (1) {
/* ❶ */  x = 1;
/* ❷ */  turn = B;
/* ❸ */  while (y && turn == B) ;
/* ❹ */  x = 0; } }

void TB() { while (1) {
/* ① */  y = 1;
/* ② */  turn = A;
/* ③ */  while (x && turn == A) ;
/* ④ */  y = 0; } }
```

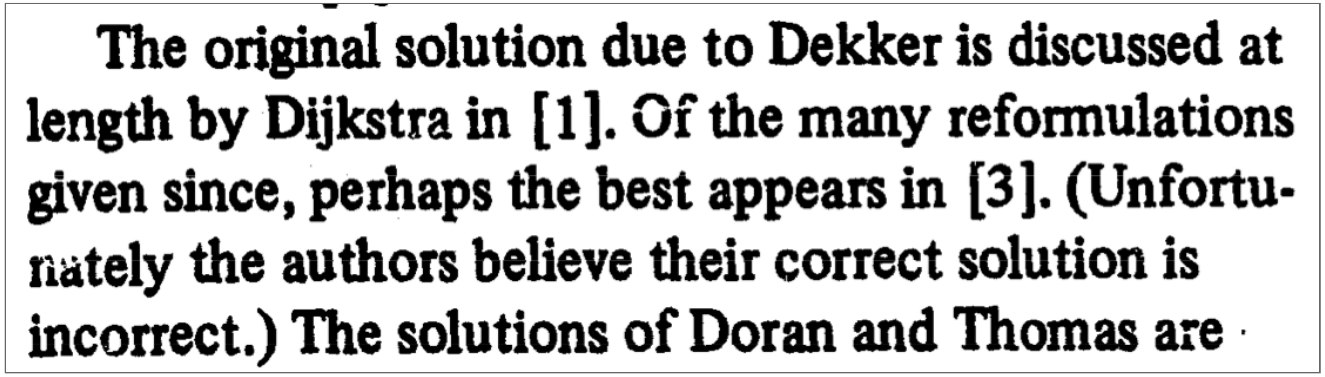
模型、模型检验与现实

“Push-button” Verification 🎖️

我们(在完全不理解算法的前提下)证明了 Sequential 内存模型下 Peterson's Protocol 的 Safety。它能够实现互斥。

并发编程比大家想象得困难

- 感受一下 **Dekker's Algorithm**
- “**Myths about the mutual exclusion problem**” (IPL, 1981)



The original solution due to Dekker is discussed at length by Dijkstra in [1]. Of the many reformulations given since, perhaps the best appears in [3]. (Unfortunately the authors believe their correct solution is incorrect.) The solutions of Doran and Thomas are

自动遍历状态空间的乐趣

可以帮助我们快速回答更多问题

- 如果结束后把门上的字条撕掉，算法还正确吗?
  - 在放下旗子之前撕
  - 在放下旗子之后撕
- 如果先贴标签再举旗，算法还正确吗?
- 我们有两个“查看”的操作
  - 看对方的旗有没有举起来
  - 看门上的贴纸是不是自己
  - 这两个操作的顺序影响算法的正确性吗?
- 是否存在“两个人谁都无法进入临界区”(liveness)、“对某一方不公平”(fairness)等行为?
  - 都转换成图 (状态空间) 上的遍历问题了!

Model Checker 和自动化

电脑为什么叫“电脑”

- 就是因为它能替代部分人类的思维活动

回忆：每个班上都有一个笔记和草稿纸都工工整整的 Ta

- 老师：布置作业画状态图
  - Ta：认认真真默默画完
    - 工整的笔记可以启发思维
    - 但 scale out 非常困难
  - 我：烦死了！劳资不干了！玩游戏去了！
    - 计算思维：写个程序 (model checker) 来辅助
      - 任何机械的思维活动都可以用计算机替代
      - AI 还可以替代启发式/经验式的决策

从模型回到现实……

回到我们的假设 (体现在模型)

- Atomic load & store
  - 读/写单个全局变量是 “原子不可分割” 的
  - 但这个假设在现代多处理器上并不成立
- 所以实际上按照模型直接写 Peterson 算法应该是错的?

“实现正确的 Peterson 算法” 是合理需求，它一定能实现

- Compiler barrier/volatile 保证不被优化的前提下
  - 处理器提供特殊指令保证可见性
  - 编译器提供 `__sync_synchronize()` 函数
    - x86: `mfence`; ARM: `dmb ish`; RISC-V: `fence rw, rw`
    - 同时含有一个 compiler barrier

🌶️ 维持实现到模型的对应

Peterson 算法：C 代码实现演示

- 一些有趣的问题
  - Compiler barrier 能够用吗?
  - 哪些地方的 barrier 是不可少的?
- 测试只能证明 “有问题”，不能证明 “没问题”

编译器到底做了什么?

- 推荐：**godbolt.org**，你不用装那些 cross compiler 了
  - 你甚至可以看到 compiler barrier 是如何在优化中传递的
    - 再一次：自动化 & 可视化的意义
  - 不懂可以把代码直接扔给 ChatGPT

原子指令

并发编程困难的解决

普通的变量读写在编译器 + 处理器的双重优化下行为变得复杂

```
retry:
  if (locked != UNLOCK) {
    goto retry;
  }
  locked = LOCK;
```

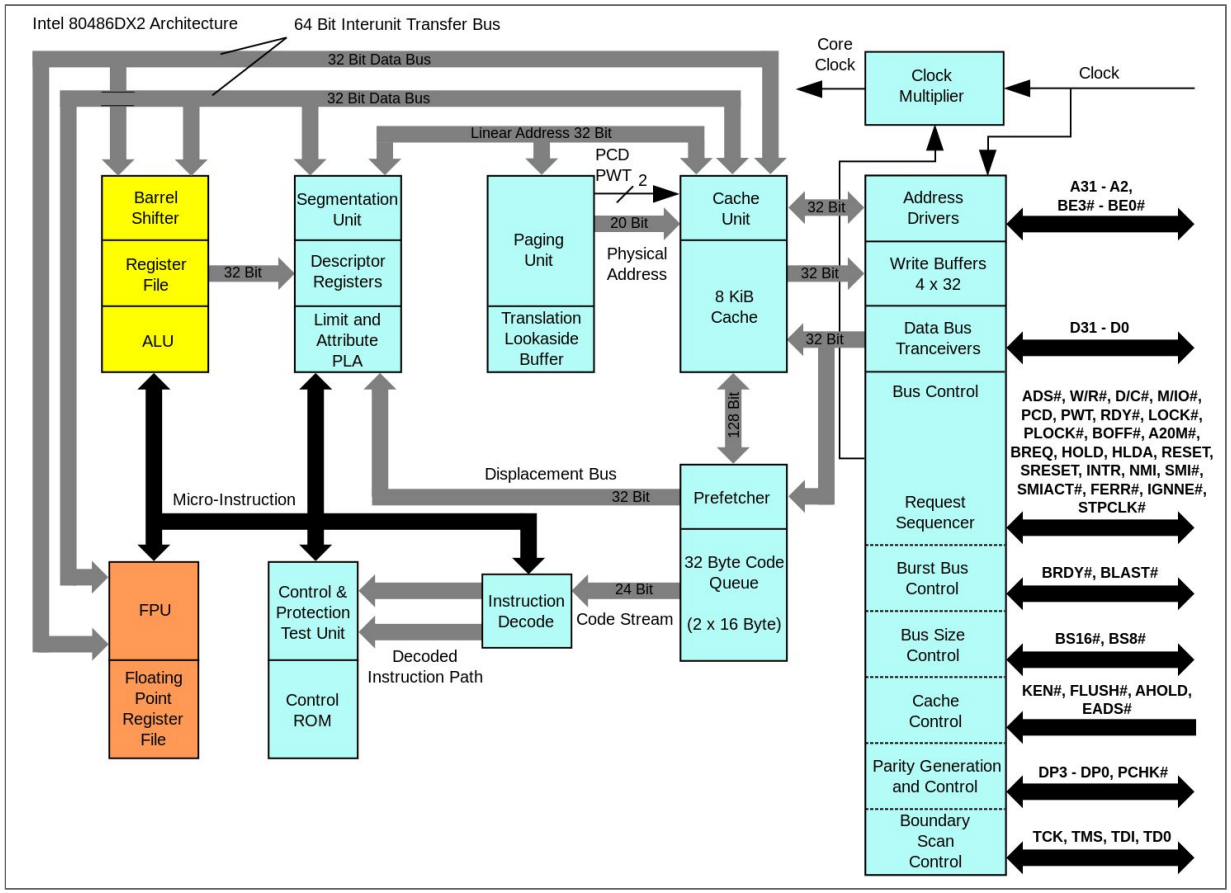
解决方法：编译器和硬件共同提供不可优化、不可打断的指令

- “原子指令” + compiler barrier

实现正确的求和

```
for (int i = 0; i < N; i++)
  asm volatile("lock incq %0" : "+m"(sum));
```

“Bus lock”——从 80386 开始引入 (bus control signal)



🌶️ 编译器和硬件的协作

Acquire/release semantics

- 对于一对配对的 release-acquire
  - (逻辑上) release 之前的 store 都对 acquire 之后的 load 可见
  - **Making Sense of Acquire-Release Semantics**
- std::atomic
  - std::memory_order_acquire: guarantees that subsequent loads are not moved before the current load or any preceding loads.
  - std::memory_order_release: preceding stores are not moved past the current store or any subsequent stores.
  - `x.load()`/`x.store()` 会根据 **memory order** 插入 fence
  - `x.fetch_add()` 将会保证 cst (sequentially consistent)
    - 去 **godbolt** 上试一下吧