Review & Comments

回顾：理解并发编程

线程 = 人

- 大脑能完成局部存储和计算

共享内存 = 物理世界

- 物理世界天生并行

一切都是状态机

- C 程序、机器指令、model checker……

互斥的目标：Stop the World

互斥：API

Safety

- 如果某个线程持有锁，则其他线程的 `lock` 不能返回
- 能**正确处理处理器乱序、宽松内存模型和编译优化**

Liveness

- 在多个线程执行 `lock` 时，至少有一个可以返回

实现互斥：自旋锁

Caveat

lock/unlock 并没有 “stop the world”

- 只是同一把锁保护的代码被串行化了
- 只有正确使用锁，才能实现 “stop the world” 的效果
  - 之后专门有一节课讲人类如何花式犯错

例子 (Data Race)

操作系统内核中的自旋锁

回顾：计算机系统的状态机模型

状态

- 共享内存 + per-CPU 寄存器

初始状态

- 由系统设计者规定 (CPU Reset)

状态迁移

- **选择任意 CPU**：
  - 从 PC 取指令执行或响应中断信号 (中断打开时)
    - 计算：改变内存/寄存器数值
    - I/O：与 “计算机系统外” 交换数据

操作系统内核中的互斥

操作系统接管了完整的计算机系统

- 每个处理器都并行 x++
- 每个处理器中断发生时执行 x += 1000000000
- (假想 x 是操作系统中的数据结构，例如进程表)

如何正确实现 x 的原子访问?

- 仅仅自旋是不够的
- 因为还有中断

正确性准则

正确实现互斥

- 关中断 + 自旋可以保证实现互斥

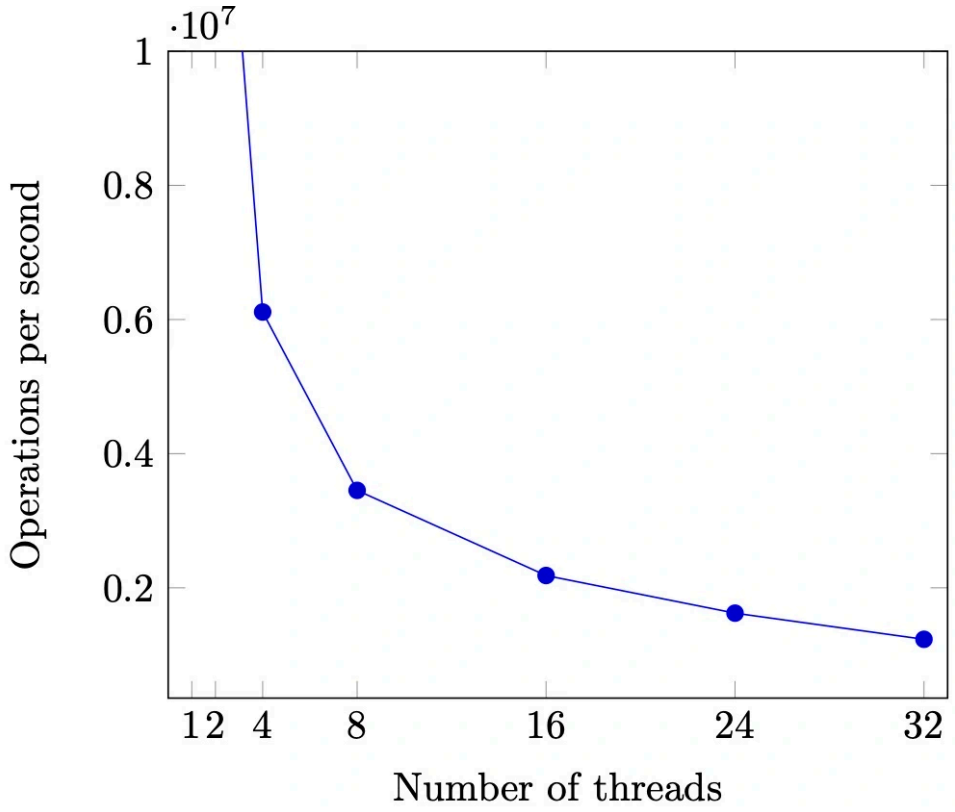
上锁/解锁前后中断状态不变

- 不得在关中断时随意打开中断 (例如处理中断时)
- 不得随意关闭中断 (否则可能导致中断丢失)
- 因此我们需要保存中断状态
  - 全局?
  - Per CPU?
  - Per Thread?
- xv6 自旋锁

Read-Copy-Update

自旋的后果

同一份计算任务，时间 (CPU cycles) 和空间 (内存占用) 会随处理器数量的增长而变化。



用自旋锁实现 sum++：更多的处理器，更差的性能

Scalability: 性能的新维度

严谨的统计很难

- CPU 动态功耗
- 系统中的其他进程
- 超线程
- NUMA
- ……

Benchmarking crimes by Gernot Heiser

自旋锁的使用场景

操作系统内核的并发数据结构 (短临界区)

- 临界区几乎不 “拥堵”，迅速结束

Kernel 里有 ~180K 个并发控制函数调用！

- 自旋锁当然不 scale
- An analysis of Linux scalability to many cores
  - 怎么办?

观察

许多操作系统内核对象具有 “read-mostly” 特点

例子

- 路由表
  - 每个数据包都要读
  - 网络拓扑改变时才变更
- 用户和组信息
  - 无时不刻在检查 (Permission Denied)
  - 但几乎从不修改用户

一个非常聪明的想法：Read-copy-update

讨论

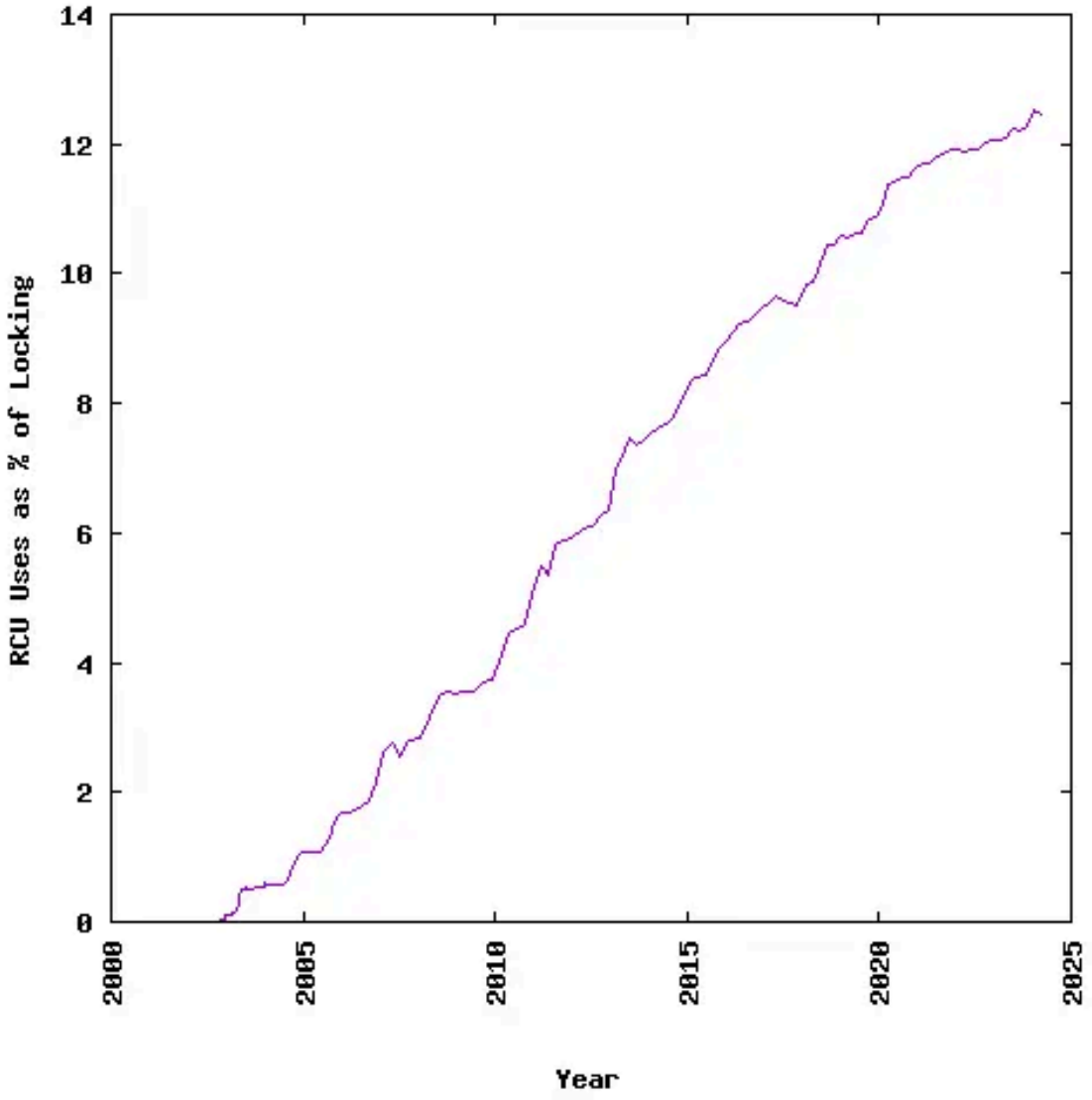
改写 = 复制

- 任何对象都可以复制！
  - (甚至可以只复制改变的部分)
  - 例子：链表
- 允许某一时刻，不同 CPU “看到” 不同版本

何时回收旧版本?

- 旧版本对象会存在一个 “graceful period”
- 直到某个时刻，所有 CPU read 都会访问到新版本
  - 怎么准确地找到这个时间点?

Linux 内核的复杂性 (不建议新手碰的原因)



应用程序和互斥锁

应用程序自旋的后果

性能问题 (1)

- 除了进入临界区的线程，其他处理器上的线程都在**空转**
  - 争抢锁的处理器越多，利用率越低
  - 如果临界区较长，不如把处理器让给其他线程

性能问题 (2)

- **应用程序不能关中断……**
  - 持有自旋锁的线程被切换
  - 导致 100% 的资源浪费
  - (如果应用程序能 “告诉” 操作系统就好了)

应用程序：实现互斥

思路：“拟人”

- 作业那么多，与其干等 Online Judge 发布，不如把自己 (CPU) 让给其他作业 (线程) 执行?

如何 “让”?

- 只有一种特殊的指令能做到：syscall
- 把锁的实现放到操作系统里就好啦
  - `syscall(SYSCALL_lock, &lk);`
    - 试图获得 `lk`，但如果失败，就切换到其他线程
  - `syscall(SYSCALL_unlock, &lk);`
    - 释放 `lk`，如果有等待锁的线程就唤醒

pthread Mutex Lock

一个足够高性能的实现

- 具有相当不错的 scalability
- 更多线程争抢时也没有极为显著的性能下降

使用方法：与自旋锁完全一致

Futex: Fast Userspace muTexes 🌶️

小孩子才做选择。操作系统当然是全都要啦！

- 性能优化的最常见技巧：**考虑平均而不是极端情况**
  - RCU 就用了这个思想！

Fast Path: 自旋一次

- 一条原子指令，成功直接进入临界区

Slow Path: 自旋失败

- 请求系统调用 `futex_wait`
- 请操作系统帮我达到自旋的效果
  - (实际上并不真的自旋)

Futex: Fast Userspace muTexes 🌶️

比你想象的复杂

- 如果没有锁的争抢，Fast Path 不能调用 `futex_wake`
- 自旋失败 → 调用 `futex_wait` → 线程睡眠
  - 如果刚开始系统调用，自旋锁被立即释放?
  - 如果任何时候都可能发生中断?

并发：水面下的冰山

- LWN: A futex overview and update
- Futexes are tricky by Ulrich Drepper