并发控制：互斥

实现互斥的根本困难：不能同时读/写共享内存.
- load 的时候不能写
  - 看到的东西马上就过时了
- store 的时候不能读
  - 也不知道把什么改成了什么

自旋锁 Spin Lock

假设硬件能为我们提供一条"瞬间完成"的读+写的指令
x86 原子操作：lock 指令前缀：处理器消灭并行 ← 硬件层面
Atomic exchange (load+store)
xchg (&x, &y)：交换 x,y 地址上的值
RISC-V 原子操作设计：
load → exec → store
Load-Reserved / Store-conditional
自旋锁的缺陷：性能问题
- 自旋(共享变量)会触发处理器间的缓存同步，延迟增加
- 除了进入临界区的线程，其它处理器上的线程都在空转 (while(1);)
  争抢锁的处理器越多，利用率越低
- 获得自旋锁的线程可能被操作系统切换出去
  - 操作系统不"感知"线程在做什么
  实现 100% 资源浪费.

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"
- 持有自旋锁时禁止执行流切换

互斥锁 Mutex Lock

Scalability：性能新维度 (可拓展性)
同一份计算任务，时间和空间会随处理器数量的增长而变化
实现线程+长临界区的互斥
- 把锁的实现放到操作系统里！
  - syscall(SYSCALL_lock, &lk);
    试图获得 lk，但如果失败，就切换到其它线程 (Sleep)
  - syscall(SYSCALL_unlock, &lk);
    释放 lk，如果有等待锁的线程就唤醒.

Futex = Spin + Mutex (Fast Userspace muTexes)

Spin Lock (线程共享 locked)
- 更快的 fast path
  - xchg 成功 → 立即进入临界区，开销很小
- 更慢的 slow path
  - xchg 失败 → 浪费 CPU 自旋等待.

Mutex Lock (通过系统调用访问 locked)
- 更快的 slow path
  - 上锁失败线程不再占有 CPU
- 更慢的 fast path
  - 即使上锁成功也需要进出内核 (syscall)

Futex
- Fast path：一条原子指令，上锁成功立即返回
- Slow path：上锁失败，执行系统调用睡眠.

先在用户空间自旋
- 如果获得锁，直接进入
- 未获得锁，系统调用
- 解锁以后也需要系统调用