并发控制：互斥

自旋锁（Spin Lock）

硬件决定哪个进程先执行

```
// 形象的代码
int table = YES;

void lock() {
retry:
  int got = xchg(&table, NOPE); // 问下系统行不行
  if (got == NOPE) // 系统不让就等着
    goto retry;
  assert(got == YES); // 系统让我做了
}

void unlock() {
  xchg(&table, YES) //做完了，把去权限还给系统
}
```

```
// 精简的代码
int locked = 0;
void lock() { while (xchg(&locked, 1)) ; }
void unlock() { xchg(&locked, 0); }
```

缺陷

性能问题 (0)

- 自旋 (共享变量) 会触发处理器间的缓存同步，延迟增加

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

性能问题 (2)

- 获得自旋锁的线程

  可能被操作系统切换出去（钥匙被带走了）

  - 操作系统不 “感知” 线程在做什么
  - (但为什么不能呢？)
- 实现 100% 的资源浪费

使用场景

使用要求

1. 临界区几乎不 “拥堵”
2. 持有自旋锁时禁止执行流切换

使用场景：操作系统内核的并发数据结构 (短临界区)

睡眠锁 (Mutex_lock)

之前的互斥锁是用 C 代码实现的，这个睡眠锁是用操作系统实现的。

实现方法：

- `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

自旋锁(Spin_lock) VS 睡眠锁(Mutex_lock)

自旋锁 (线程直接共享 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

睡眠锁 (通过系统调用访问 locked)

- 更快的 slow path
  - 上锁失败线程不再占用 CPU
- 更慢的 fast path
  - 即便上锁成功也需要进出内核 (syscall)

互斥锁（Futex_lock）

由之前的比较得到改进的想法：

- 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