第6讲 并发控制基础（Peterson算法、原子操作）

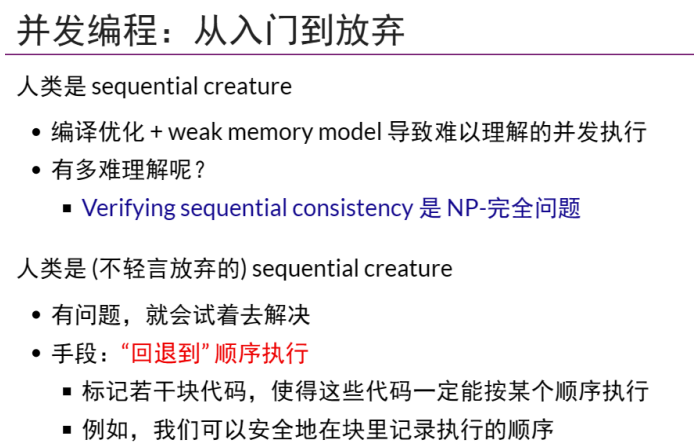
1. 并发编程为什么困难

> 本讲其实讲的东西和达达讲的事务那边都很像啊！逻辑上来说确实是啊！两个线程和两个事务，都是一起在做一件事情！我们希望两个东西执行的“结果”看上去和“顺序执行”是一致的！

背景回顾：虽然 “线程库” 入门简单，但多处理器编程 + 编译优化会给我们带来很多意想不到的惊喜。在编写多线程程序时，们必须放弃许多对顺序程序编程时的基本假设，这也是并发编程困难的原因。

本讲内容：并发编程困难不代表我们只能摆烂——我们还可以创造出新的手段，帮助我们编写正确的并发程序：

- 互斥问题和 Peterson 算法
- Peterson 算法的正确性和模型检验
- Peterson 算法在现代多处理器系统上的实现
- 实现并发控制的硬件和编译器机制



2. 互斥和失败的尝试

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

但单纯这样做，多处理器就毫无意义！

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

就好像上厕所看门上有没有锁一样。

代码不正确的原因？在物理世界里面也会发生。因为“人很大”，人相对于原子的大小比例很大。所以人会假设两个人在时间尺度上面拿锁的时候能够做一个协调。所以拿锁的时候，它不能保证我看到那里有lock的时候，我伸手去拿，我还能拿到它。

3. 原子指令

为什么这样一个很简单、很符合我们直觉的程序无法在现代处理器上工作？

因为在人类的世界中，你是一边看一边拿的，但这件事情在计算机世界里做不到。所以我们需要硬件和编译器来帮助我们实现这一点！（同时禁止编译优化）

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

总结

并发编程 “很难”：想要完全理解并发程序的行为，是非常困难的——我们甚至可以利用一个 “万能” 的调度器去帮助我们求解 NP-完全问题。因此，人类应对这种复杂性的方法就是退回到不并发。通过互斥实现 stop/resume the world，我们就可以使并发程序的执行变得更容易理解——而只要程序中 “能并行” 的部分足够多，串行化一小部分也并不会对性能带来致命的影响。