并发控制：使用信号量的同步 notes

速通：*Operating Systems: Three Easy Pieces*, Chapter 31 – Semaphores

信号量：条件变量的特例

```c
void P(sem_t *sem) { // wait
  wait_until(sem->count > 0) {
    sem->count--;
  }
}

void V(sem_t *sem) { // post (signal)
  sem->count++;
}
```

因为条件的特殊性，信号量不需要 broadcast。

- P 失败时立即睡眠等待
- 执行 V 时，唤醒任意等待的线程

理解信号量

考虑 count=1 时，那么 P 和 V 就成为互斥锁。

```c
#define YES 1
#define NO 0

void lock() {
  wait_until(count == YES) {
    count = NO;
  }
}

void unlock() {
```

```
  count = YES;
}
```

P(prolaag) – try + decrease/down/wait/acquire

- 试着从袋子里拿一个球
  - 如果拿到了，离开
  - 如果袋子空了，排队等待

V(verhoog) – increase/up/post/signal/release

- 往袋子里放一个球
  - 如果有人在等球，他就可以拿走刚放进去的球了
  - 放球–拿球的过程实现了同步

扩展的互斥锁：信号量对应“资源数量”

信号量：实现优雅的生产者–消费者模型

信号量设计的重点

- 考虑每一个单元的资源是什么
- 生产者/消费者：把球从一个袋子放到另一个袋子里

```c
void Tproduce() {
  P(&empty);
  printf("("); // 注意共享数据结构访问需互斥
  V(&fill);
}
void Tconsume() {
  P(&fill);
  printf(")");
  V(&empty);
}
```

Take-away Messages

信号量是一种特殊的条件变量，而且可以在操作系统上被高效地实现，避免 broadcast 唤醒的浪费：

```c
void P() {
  WAIT_UNTIL(count > 0) {
    count--;
  }
}
void V() {
  count++;
}
```

同时，我们也可以把信号量理解成袋子里的球，或是管理游泳池的手环，因此它在符合这个抽象时，能够带来优雅的代码。

更重要的是，但凡我们能将任务很好地分解成少量串行的部分和绝大部分“线程局部”的计算，那么生产者–消费者和计算图模型就能实现有效的并行。精心设计的分布式同步协议不仅可能存在正确性漏洞，带来的性能收益很可能也是微乎其微的。