同步问题

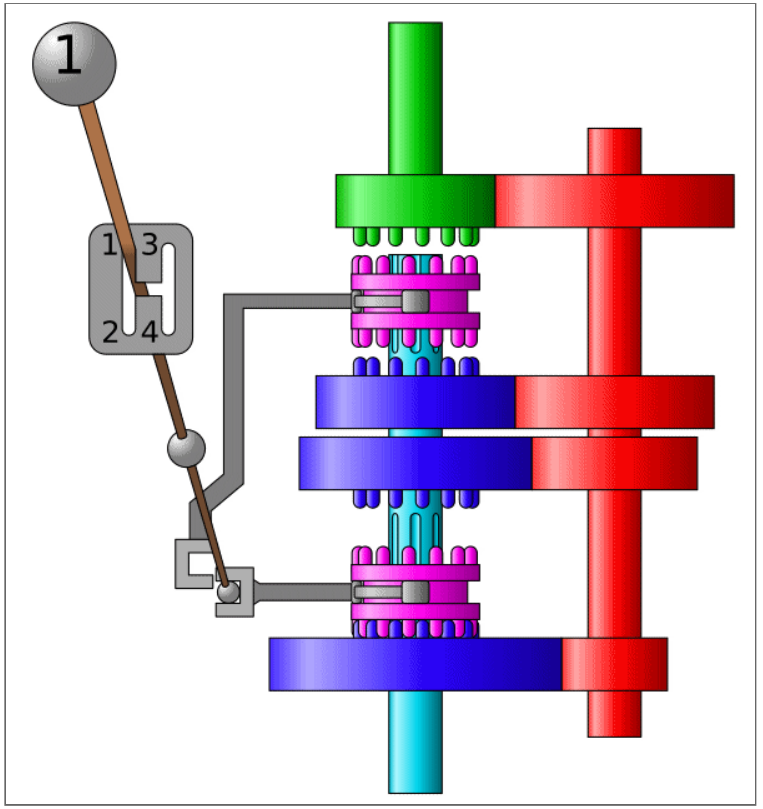
同步 (Synchronization)

两个或两个以上随时间变化的量在变化过程中保持一定的相对关系

- 同步电路 (一个时钟控制所有触发器)
- iPhone/iCloud 同步 (手机 vs 电脑 vs 云端)
- 变速箱同步器 (合并快慢速齿轮)
- 同步电机 (转子与磁场转速一致)
- 同步电路 (所有触发器在边沿同时触发)

异步 (Asynchronous) = 不需要同步

- 上述很多例子都有异步版本 (异步电机、异步电路、异步线程)



并发程序中的同步

并发程序的步调很难保持“完全一致”

- 线程同步：在某个时间点共同达到互相已知的状态

再次把线程想象成我们自己

- NPY：等我洗个头就出门/等我打完这局游戏就来
- 舍友：等我修好这个 bug 就吃饭
- 导师：等我出差回来就讨论这个课题
- jyy: ~~等我成为卷王就躺平~~
  - “先到先等”，在条件达成的瞬间再次恢复并行
  - 同时开始出去玩/吃饭/讨论

生产者-消费者问题：学废你就赢了

99% 的实际并发问题都可以用生产者-消费者解决。

```
void Tproduce() { while (1) printf("("); }
void Tconsume() { while (1) printf(")"); }
```

在 printf 前后增加代码，使得打印的括号序列满足

- 一定是某个合法括号序列的前缀
- 括号嵌套的深度不超过 n
  - $n = 3$, (((())())(((合法
  - $n = 3$, ((((())))), ((())) 不合法
- 生产者-消费者问题中的同步
  - Tproduce: 等到有空位时才能打印左括号
  - Tconsume: 等到有多余的左括号时才能打印右括号

计算图、调度器和生产者-消费者问题

为什么叫“生产者-消费者”而不是“括号问题”?

- 左括号：生产资源 (任务)、放入队列
- 右括号：从队列取出资源 (任务) 执行

并行计算基础：计算图

- 计算任务构成有向无环图
  - $(u, v) \in E$ 表示 v 要用到前 u 的值
- 只要调度器 (生产者) 分配任务效率够高，算法就能并行
  - 生产者把任务放入队列中
  - 消费者 (workers) 从队列中取出任务

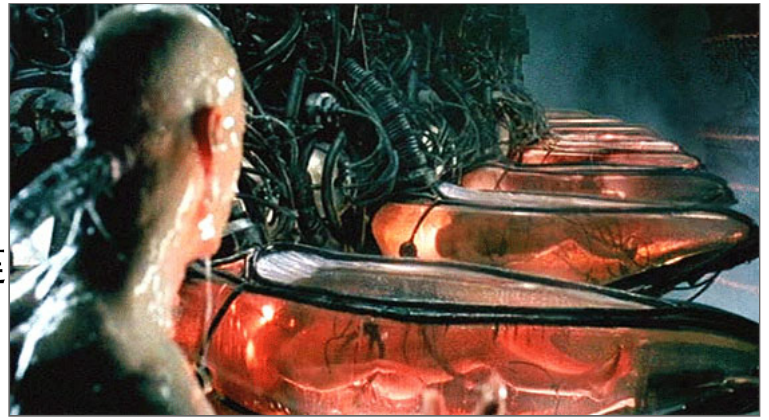
生产者-消费者：实现

能否用互斥锁实现括号问题?

- 左括号：嵌套深度 (队列) 不足 n 时才能打印
- 右括号：嵌套深度 (队列) > 1 时才能打印
  - 当然是等到满足条件时再打印了 (代码演示)
    - 用互斥锁保持条件成立

并发：小心!

- 压力测试 + 观察输出结果
- 自动观察输出结果：pc-check.py
- 未来：copilot 观察输出结果，并给出修复建议
- 更远的未来：~~我们都不需要不存在子~~



条件变量

同步问题：分析

线程同步由条件不成立等待和同步条件达成继续构成

线程 join

- Tmain 同步条件：`nexit == T`
- Tmain 达成同步：最后一个线程退出 `nexit++`

生产者/消费者问题

- Tproduce 同步条件：`CAN_PRODUCE（count < n）`
- Tproduce 达成同步：Tconsume `count--`
- Tconsume 同步条件：`CAN_CONSUME（count > 0）`
- Tconsume 达成同步：Tproduce `count++`

理想中的同步 API

```
wait_until(CAN_PRODUCE) {
  count++;
  printf("(");
}

wait_until(CAN_CONSUME) {
  count--;
  printf(")");
}
```

若干实现上的难题

- 正确性
  - 大括号内代码执行时，其他线程不得破坏等待的条件
- 性能
  - 不能 spin check 条件达成
  - 已经在等待的线程怎么知道条件被满足?

条件变量：理想与实现之间的折衷

一把互斥锁 + 一个“条件变量” + 手工唤醒

- wait(cv, mutex) 💤
  - 调用时必须保证已经获得 mutex
  - wait 释放 mutex、进入睡眠状态
  - 被唤醒后需要重新执行 lock(mutex)
- signal/notify(cv) 💬
  - 随机私信一个等待者：醒醒
  - 如果有线程正在等待 cv，则唤醒其中一个线程
- broadcast/notifyAll(cv) 📣
  - 叫醒所有人
  - 唤醒全部正在等待 cv 的线程

条件变量：实现生产者-消费者

```
void Tproduce() {
  mutex_lock(&lk);
  if (!CAN_PRODUCE) cond_wait(&cv, &lk);
  printf("("); count++; cond_signal(&cv);
  mutex_unlock(&lk);
}

void Tconsume() {
  mutex_lock(&lk);
  if (!CAN_CONSUME) cond_wait(&cv, &lk);
  printf(")"); count--; cond_signal(&cv);
  mutex_unlock(&lk);
}
```

代码演示 & 压力测试 & 模型检验

- (Small scope hypothesis)

条件变量：正确的打开方式

同步的本质：`wait_until(COND) { ... }`，因此：

- 需要等待条件满足时

```
mutex_lock(&mutex);
while (!COND) {
  wait(&cv, &mutex);
}
assert(cond);  // 互斥锁保证条件成立
mutex_unlock(&mutex);
```

- 任何改动使其他线可能被满足时

```
mutex_lock(&mutex);
// 任何可能使条件满足的代码
broadcast(&cv);
mutex_unlock(&mutex);
```

条件变量：应用

条件变量：万能并行计算框架 (M2)

```
struct work {
  void (*run)(void *arg);
  void *arg;
}

void Tworker() {
  while (1) {
    struct work *work;
    wait_until(has_new_work() || all_done) {
      work = get_work();
    }
    if (!work) break;
    else {
      work->run(work->arg); // 允许生成新的 work（注意互斥）
      release(work);  // 注意回收 work 分配的资源
    }
```

```
    }
  }
```

条件变量：更古怪的习题/面试题

有三种线程

- Ta 若干: 死循环打印 <
- Tb 若干: 死循环打印 >
- Tc 若干: 死循环打印 _

任务：

- 对这些线程进行同步，使得屏幕打印出 <><_ 和 ><>_ 的组合

使用条件变量，只要回答三个问题：

- 打印 “<” 的条件?
- 打印 “>” 的条件?
- 打印 “_” 的条件?