

同步

我们已经了解如何通过 “不可优化、保证顺序” 的原子指令实现自旋锁，以及借助操作系统（系统调用）实现线程的睡眠，从而不致于出现 CPU 空转的浪费。

目录

- 同步问题的定义
- 生产者-消费者问题
- 条件变量

同步问题的定义

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

生产者-消费者问题

99% 的实际并发问题都可以用生产者-消费者解决。

```
1 void Tproduce() {
2     while (1)
3         printf("(");
4 }
5
6 void Tconsume() {
7     while (1)
8         printf(")");
9 }
```

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



```
1 def wait_until(CAN_PRODUCE):
2     count++
3     printf("")
4
5 def wait_until(CAN_CONSUME):
6     count--
7     print("")
```

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/notify All(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

```

1 void Tproduce() {
2     mutex_lock(&lk);
3     if (!CAN_PRODUCE) cond_wait(&cv, &lk);
4     printf("("); count++; cond_signal(&cv);
5     mutex_unlock(&lk);
6 }
7
8 void Tconsume() {
9     mutex_lock(&lk);
10    if (!CAN_CONSUME) cond_wait(&cv, &lk);
11    printf(")"); count--; cond_signal(&cv);
12    mutex_unlock(&lk);
13 }

```

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

```

1 mutex_lock(&mutex);
2 while (!COND) {
3     wait(&cv, &mutex);
4 }
5 assert(cond); // 互斥锁保证条件成立
6 mutex_unlock(&mutex);

```

- 任何改动使其他线程可能被满足时

```

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

```

条件变量：应用

条件变量：万能并行计算框架（M2）

```
1 struct work {
2     void (*run)(void *arg);
3     void *arg;
4 }
5
6 void Tworker() {
7     while (1) {
8         struct work *work;
9         wait_until(has_new_work() || all_done) {
10             work = get_work();
11         }
12         if (!work) break;
13         else {
14             work->run(work->arg); // 允许生成新的 work (注意互斥)
15             release(work); // 注意回收 work 分配的资源
16         }
17     }
18 }
```

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？

Take-away Messages

同步的本质是线程需要等待某件它所预期的事件发生，而事件的发生总是可以用共享状态的条件来表达。并且在这个条件被满足的前提下完成一些动作：

```
1 WAIT_UNTIL(cond) with (mutex) {  
2     // cond 在此时成立  
3     work();  
4 }
```

计算机系统的设计者提供了条件变量的机制模仿这个过程，它与互斥锁联合使用：

- `cond_wait(cv, lk)` 释放互斥锁 `lk` 并进入睡眠状态。注意被唤醒时，`cond_wait` 会重新试图获得互斥，直到获得互斥锁后才能返回。
- `cond_signal(cv)` 唤醒一个在 `cv` 上等待的线程
- `cond_broadcast(cv)` 唤醒所有在 `cv` 上等待的线程

我们也很自然地可以用 `wait + broadcast` 实现 `WAIT_UNTIL`，从而实现线程之间的同步。

课后习题/编程作业

1. 阅读材料

教科书 [Operating Systems: Three Easy Pieces](#):

- 第 29 章 - Locked Data Structures
- 第 30 章 - Condition Variables

教科书详细解释了条件变量错误使用的案例。因为条件变量自身设计的缺陷（库函数希望 API 保持简单，而不是唤醒“满足某些条件的线程”），就带来了 `cond_signal` 使用上的不便。作为折衷，我们推荐 `cond_broadcast` 的“万能”同步方法。

2. 编程实践

运行示例代码并观察执行结果。建议大家先不参照参考书，通过两个信号量分别代表 `Tproduce` 和 `Tconsume` 的唤醒条件实现同步。

3. 实验作业

开始 M2 和 L1 实验作业。

信号量

我们分析了同步的本质需求：两个并发的线程等待某个同步条件达成，完成时间线的“交汇”。相应地，我们有了条件变量实现同步，并且解决了生产者-消费者问题（括号打印问题）。

本讲内容：另一种共享内存系统中常用的同步方法：信号量（E. W. Dijkstra）

目录

- 什么是信号量
- 信号量适合解决什么问题
- 哲♂学家吃饭问题