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

信号量

复习：生产者-消费者、互斥、条件变量

打印“合法”的括号序列 `(())()`

- 左括号对应 push
- 右括号对应 pop

```
#define CAN_PRODUCE (count < n)
#define CAN_CONSUME (count > 0)

wait_until(CAN_PRODUCE)
  with (mutex) {
    count++;
    printf("(");
  }

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

信号量：一种条件变量的特例

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

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

正是因为条件的特殊性，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（1）

初始时 count = 1 的特殊情况

- 互斥锁是信号量的特例

```
#define YES 1
#define NO 0

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

void unlock() {
  count = YES;
}
```

理解信号量（2）

`P - prolaag (try + decrease/down/wait/acquire)`

- 试着从袋子里取一个球
  - 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

理解信号量（3）

扩展的互斥锁：一个手环 → *n* 个手环

- 让更多同学可以进入更衣室
  - 管理员可以持有任意数量的手环（count，更衣室容量上限）
  - 先进入更衣室的同学先进入游泳池
  - 手环用完后需要等同学出来
- 信号量对应了“资源数量”

信号量：实现优雅的生产者-消费者

信号量设计的重点

- 考虑“球”/“手环”（每一单位的“资源”）是什么
- 生产者/消费者 = 把球从一个袋子里放到另一个袋子里

```
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);
}
```

信号量：应用

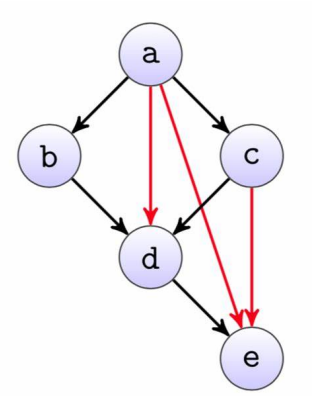
信号量的两种典型应用

1. 实现一次临时的 happens-before
    - 初始：s = 0
    - A; V(s)
    - P(s); B
        - 假设 s 只被使用一次，保证 A happens-before B
2. 实现计数型的同步
    - 初始：done = 0
    - Tworker：V(done)
    - Tmain：P(done) ××T

对应了两种线程 join 的方法

- 1→2→…$T1$→$T2$→… v.s. 完成就行，不管顺序

例子：实现计算图



对于任何计算图

- 为每个节点分配一个线程
    - 对每条入边执行 P（wait）操作
    - 完成计算任务
    - 对每条出边执行 V（post/signal）操作
        - 每条边恰好 P 一次、V 一次
        - PLCS 直接就解决了啊？

```
void Tworker_d() {
  P(bd); P(ad); P(cd);
  // 完成节点 d 上的计算任务
  V(de);
}
```

实现计算图（cont'd）

乍一看很厉害

- 完美解决了并行问题

实际上……

- 创建那么多线程和那么多信号量 = Time Limit Exceeded
- 解决线程太多的问题
  - 一个线程负责多个节点的计算
    - 静态划分 → 覆盖问题
    - 动态调度 → 又变回了生产者-消费者
- 解决信号量太多的问题
  - 计算节点共享信号量
    - 可能出现“假唤醒” → 又变回了条件变量

例子：毫无意义的练习题

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 `_`
- 如何同步这些线程，保证打印出 `<><_` 和 `><>_` 的序列？

信号量的困难

- 上一条鱼打印后，`<` 和 `>` 都是可行的
- 我应该 P 哪个信号量？
  - 可以 P 我自己的
  - 由打印 `_` 的线程随机选一个

例子：使用信号量实现条件变量

```
semaphore mutex = 1;
semaphore cond = 0;
int count=0;
```

```

void wait() {
  P(&mutex);
  count++;
  if (count == 1) {
    P(&cond);
  }
  V(&mutex);
  P(&cond);
  P(&mutex);
  count--;

  if (count == 0) {
    V(&cond);
  }

  V(&mutex);
}
```

```
void signal() {
  P(&mutex);
  if (count == 0) {
    V(&cond);
  }

  count++;
  V(&mutex);
}
```

```
void broadcast() {
  P(&mutex);
  while(count > 0) {
    V(&cond);
    count--;
  }
  V(&mutex);
}0
```

- New Bing 给出了一种“思路”
  - 第一个 wait 的线程会在持有 mutex 的情况下 P(cond)
  - 从此再也没有人能获得互斥锁……
    - ~~像极了我改期末试卷的体验~~

使用信号量实现条件变量：本质困难

操作系统用自旋锁保证 wait 的原子性

```
wait(cv, mutex) {
  release(mutex);
  sleep();
}
```

信号量实现的矛盾

- 不能带着锁睡眠（NewBing 犯的错误）
- 也不能先释放锁
  - `P(mutex); nwait++; V(mutex);`
  - 此时 signal/broadcast 发生，唤醒了后 wait 的线程
  - `P(sleep);`
- （我们稍后介绍解决这种矛盾的方法）

信号量的使用：小结

信号量是对“袋子和球/手环”的抽象

- 实现一次 happens-before，或是计数型的同步
  - 能够写出优雅的代码
  - `P(empty); printf("("); V(fill)`
- 但并不是所有的同步条件都容易用这个抽象来表达

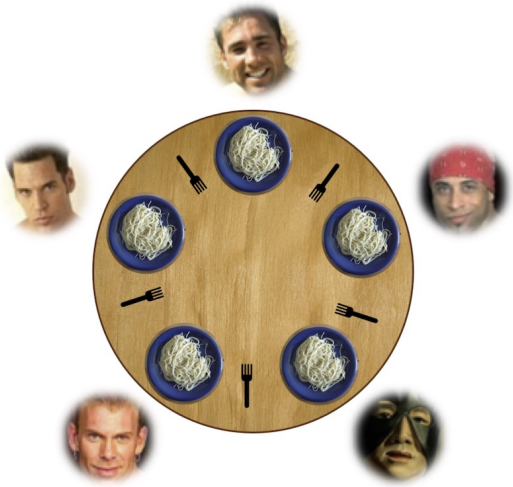
> TIP: BE CAREFUL WITH GENERALIZATION
>
> The abstract technique of generalization can thus be quite useful in systems design, where one good idea can be made slightly broader and thus solve a larger class of problems. However, be careful when generalizing; as Lampson warns us “Don’t generalize; generalizations are generally wrong” [L83].
>
> One could view semaphores as a generalization of locks and condition variables; however, is such a generalization needed? And, given the difficulty of realizing a condition variable on top of a semaphore, perhaps this generalization is not as general as you might think.

哲♂学家吃饭问题

哲学家吃饭问题（E. W. Dijkstra, 1960）

经典同步问题：哲学家（线程）有时思考，有时吃饭

- 吃饭需要同时得到左手和右手的叉子
- 当叉子被其他人占有时，必须等待，如何完成同步？



失败与成功的尝试

失败的尝试

- 把信号量当互斥锁：先拿一把叉子，再拿另一把叉子

成功的尝试（万能的方法）

```
#define CAN_EAT (avail[lhs] && avail[rhs])
mutex_lock(&mutex);
while (!CAN_EAT)
  cond_wait(&cv, &mutex);
avail[lhs] = avail[rhs] = false;
mutex_unlock(&mutex);

mutex_lock(&mutex);
avail[lhs] = avail[rhs] = true;
cond_broadcast(&cv);
mutex_unlock(&mutex);
```

成功的尝试：信号量

Trick: 死锁会在 5 个哲学家“同时吃饭”时发生

- 破坏这个条件即可
  - 保证任何时候至多只有 4 个人可以吃饭

    - 直观理解：大家先从桌上退出
        - 袋子里有 4 张卡
        - 拿到卡的可以上桌吃饭（拿叉子）
        - 吃完以后把卡归还到袋子
- 任意 4 个人想吃饭，总有一个可以拿起左右手的叉子
    - 教科书上有另一种解决方法（lock ordering；之后会讲）

但这真的对吗？

- philosopher-check.py

反思：分布与集中

“Leader/follower” - 有一个集中的“总控”，而非“各自协调”

- 在可靠的消息机制上实现任务分派
    - Leader 串行处理所有请求（例如：条件变量服务）

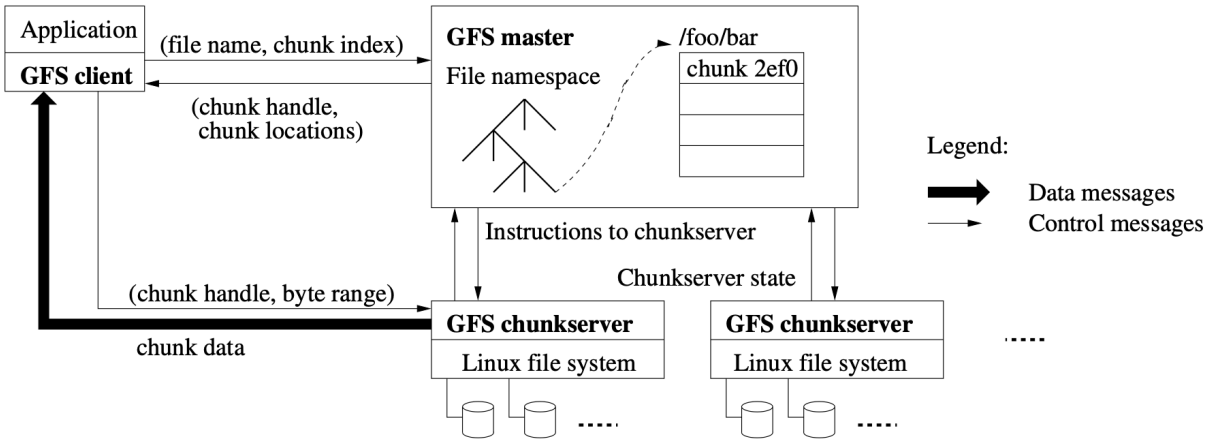
```
void Tphilosopher(int id) {
  send(Twaiter, id, EAT);
  receive(Twatier); // 等待 waiter 把两把叉子递给哲学家
  eat();
  send(Twaiter, id, DONE); // 归还叉子
}

void Twaiter() {
  while (1) {
    (id, status) = receive(Any);
    switch (status) { ... }
  }
}
```

反思：分布与集中（cont'd）

你可能会觉得，管叉子的人是性能瓶颈

- 一大桌人吃饭，每个人都叫服务员的感觉
- Premature optimization is the root of all evil (D. E. Knuth)



抛开 workload 谈优化就是耍流氓

- 吃饭的时间通常远远大于请求服务员的时间
- 如果一个 manager 搞不定，可以分多个（fast/slow path）
  - 把系统设计好，集中管理可以不是瓶颈：The Google File System（SOSP'03）开启大数据时代

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：

```
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。

课后习题/编程作业

1. 阅读材料

教科书 Operating Systems: Three Easy Pieces：

- 第 31 章 - Semaphores

2. 编程实践

运行示例代码并观察执行结果，有兴趣的同学可以试着用信号量实现条件变量——当然，仅仅有一个实现是不够的，你应当尽可能地压力测试它。