

并发控制：同步

蒋炎岩

南京大学



计算机科学与技术系



计算机软件研究所



Overview

复习

- 互斥：自旋锁、互斥锁、futex
- ~~是时候面对真正的并发编程了~~

本次课回答的问题

- Q: 如何在多处理器上协同多个线程完成任务？

本次课主要内容

- 典型的同步问题：生产者-消费者；哲学家吃饭
- 同步的实现方法：信号量、条件变量

线程同步

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

- 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 (转子与磁场速度一致)
- 同步电路 (所有触发器在边沿同时触发)

异步 (Asynchronous) = 不同步

- 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”



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
- 同步
 - 等到有空位再打印左括号
 - 等到能配对时再打印右括号

生产者-消费者问题：分析

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

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： pc.c
 - 用互斥锁保持条件成立
 - 压力测试的检查当然不能少： pc-check.py
 - Model checker 当然也不能少 (留作习题)

条件变量：万能同步方法

同步问题：分析

任何同步问题都有**先来先等待**的条件。

线程 join (thread.h, sum.c)

- 等所有线程结束后继续执行，否则等待

NPY 的例子

- 打完游戏且洗完头后继续执行 `date()`，否则等待

生产者/消费者问题




- 左括号：深度 $k < n$ 时 `printf`，否则等待
- 右括号： $k > 0$ 时 `printf`，否则等待
 - 再看一眼 pc.c

Conditional Variables (条件变量, CV)

把 pc.c 中的自旋变成睡眠

- 在完成操作时唤醒


条件变量 API

- wait(cv, mutex) 
 - 调用时必须保证已经获得 mutex
 - 释放 mutex、进入睡眠状态
- signal/notify(cv)  私信：走起
 - 如果有线程正在等待 cv，则唤醒其中一个线程
- broadcast/notifyAll(cv)  所有人：走起
 - 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 (count == n) cond_wait(&cv, &lk  
    printf("("); count++; cond_signal(  
    mutex_unlock(&lk);  
}
```

```
void Tconsume() {  
    mutex_lock(&lk);  
    if (count == 0) cond_wait(&cv, &lk  
    printf(")"); count--; cond_signal(  
    mutex_unlock(&lk);  
}
```



压力测试： **pc-cv.c**； 模型检验： **pc-cv.py**

- (Small scope hypothesis)

条件变量：正确的打开方式

需要等待条件满足时

```
mutex_lock(&mutex);  
while (!cond) {  
    wait(&cv, &mutex);  
}  
assert(cond);  
// ...  
// 互斥锁保证了在此期间条件 cond 总是为真  
// ...  
mutex_unlock(&mutex);
```

其他线程条件可能被满足时

```
broadcast(&cv);
```

- 修改 pc-cv.c 和 pc-cv.py

条件变量：实现并行计算

```
struct job {  
    void (*run)(void *arg);  
    void *arg;  
}  
  
while (1) {  
    struct job *job;  
  
    mutex_lock(&mutex);  
    while (! (job = get_job())) {  
        wait(&cv, &mutex);  
    }  
    mutex_unlock(&mutex);  
  
    job->run(job->arg); // 不需要持有锁  
                        // 可以生成新的任务  
                        // 注意回收分配  
}
```

条件变量：更古怪的习题/面试题

有三种线程，分别打印 <, >, 和 _

- 对这些线程进行同步，使得打印出的序列总是 <><_ 和 ><>_ 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？
 - fish.c

信号量

复习：互斥锁和更衣室管理

操作系统 = 更衣室管理员

- 先到的人 (线程)
 - 成功获得手环，进入游泳馆
 - $*lk = \text{🔒}$ ，系统调用直接返回
- 后到的人 (线程)
 - 不能进入游泳馆，排队等待
 - 线程放入等待队列，执行线程切换 (yield)
- 洗完澡出来的人 (线程)
 - 交还手环给管理员；管理员把手环再交给排队的人
 - 如果等待队列不空，从等待队列中取出一个线程允许执行
 - 如果等待队列为空， $*lk = \text{✅}$
- 管理员 (OS) 使用自旋锁确保自己处理手环的过程是原子的



更衣室管理

完全没有必要限制手环的数量——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 (更衣室容量上限)
 - 先进入更衣室的同学先得到
 - 手环用完后才需要等同学出来



V.S.



更衣室管理 (by E.W. Dijkstra)

做一点扩展——线程
可以任意“变出”一个
手环



- 把手环看成是令牌
- 得到令牌的可以进入执行
- 可以随时创建令牌

“手环” = “令牌” = “一个资源” = “信号量”
(semaphore)

- $P(&sem) - \text{prolaag} = \text{try} + \text{decrease}; \text{wait}; \text{down}; \text{in}$
 - 等待一个手环后返回
 - 如果此时管理员手上有空闲的手环，立即返回
- $V(&sem) - \text{verhoog} = \text{increase}; \text{post}; \text{up}; \text{out}$
 - 变出一个手环，送给管理员
- 信号量的行为建模: [sem.py](#)

信号量：实现生产者-消费者

信号量设计的重点

- 考虑“手环”（每一单位的“资源”）是什么，谁创造？谁获取？

- pc-sem.c

```
void producer() {  
    P(&empty);    // P()返回 -> 得到手环  
    printf("(");  // 假设线程安全  
    V(&fill);  
}  
  
void consumer() {  
    P(&fill);  
    printf(")");  
    V(&empty);  
}
```

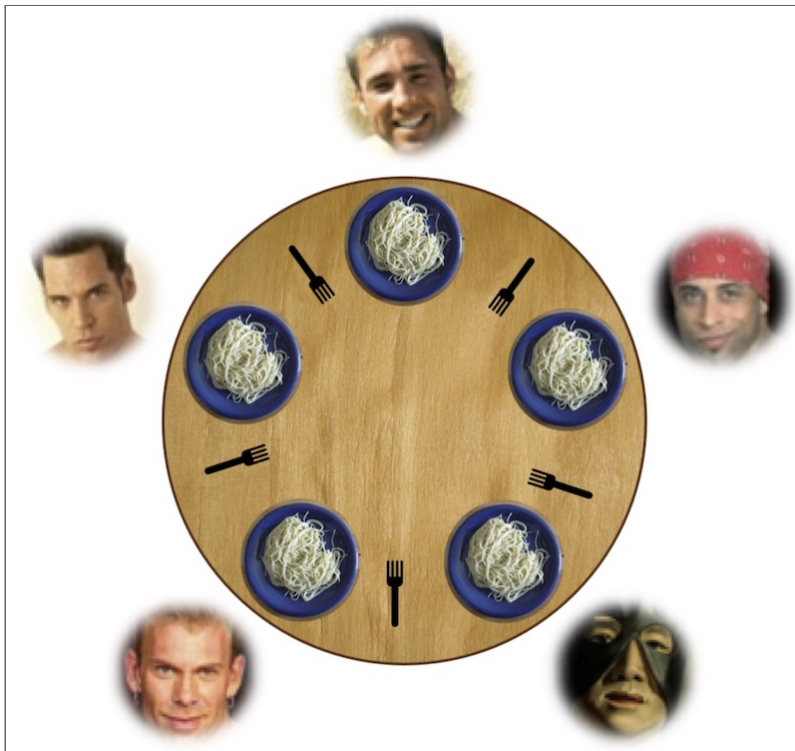
- 在“一单位资源”明确的问题上更好用

哲学家吃饭问题

哲学家吃饭问题 (E. W. Dijkstra, 1960)

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？
 - 如何用互斥锁/信号量实现？



失败与成功的尝试

失败的尝试

- **philosopher.c** (如何解决?)

成功的尝试 (万能的方法)

```
mutex_lock(&mutex);  
while (!(avail[lhs] && avail[rhs]))  
    wait(&cv, &mutex);  
}  
avail[lhs] = avail[rhs] = false;  
mutex_unlock(&mutex);  
  
mutex_lock(&mutex);  
avail[lhs] = avail[rhs] = true;  
broadcast(&cv);  
mutex_unlock(&mutex);
```




忘了信号量，让一个人集中管理叉子吧！

“Leader/follower” - 生产者/消费者

- 分布式系统中非常常见的解决思路 (HDFS, ...)

```
void Tphilosopher(int id) {
    send_request(id, EAT);
    P(allowed[id]); // waiter 会把叉子
    philosopher_eat();
    send_request(id, DONE);
}

void Twaiter() {
    while (1) {
        (id, status) = receive_request()
        if (status == EAT) { ... }
        if (status == DONE) { ... }
    }
}
```



忘了那些复杂的同步算法吧！

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 谈优化就是要流氓

- 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，使集中管理不成为瓶颈
 - Millions of tiny databases (NSDI'20)

总结

总结

本次课回答的问题

- Q: 如何在多处理器上协同多个线程完成任务？

Take-away message

- 实现同步的方法
 - 条件变量、信号量；生产者-消费者问题
 - Job queue 可以实现几乎任何并行算法
- 不要“自作聪明”设计算法，小心求证

End.