

SE-227

那么我们开始复习 CSE 下半学期的内容吧。

需要复习的内容主要是 Lecture 14 ~ 28。实际上最後两 Lecture 主要用于复习，所以我们还是从现在开始瞧一瞧。

Lecture 14 & 15: AoN & BoA

赌博有个常用术语：Double or Nothing。意思是参加一场赌局，有可能获得 100% 收益，即拿到 100% 赌资的回报，要么全部输光全部赌资。

本节的主要内容也就是 All or Nothing & Before or After。

下面我们详细讲讲。

The CAP Theory

所谓卡普理论……是什么意思？

也是一个类似不可能三角的理论：无法同时保证「一致性」、「可用性」，「区域性网络故障容忍性」三项；最多只能保证其中两项。

- 2010 年由 UC Berkeley 的 Eric Brewer 教授首次提出
- 2012 年由 MIT 的 Seth Gilbert 和 Nancy lynch 证明

更形式化一点说，对分布式计算机系统来说，同时满足以下三点是不可能的：

- Consistency、Availability、Partition Tolerance 不可得兼。

这个就是简称 CAP Theorem。

Consistency

何谓一致性？令所有节点在同样的时间看见同样的数据。即使在数据有多处备份的情况下依然要保证。

Availability

定义为系统对用户请求均有响应，要能在一定时间内回复处理成功或失败。不能在「不确定」状态上停留太久。

Partition Tolerance

这个可能比较抽象：在任意消息丢失或一部分的系统故障情况下，系统依然可继续运行。

| 事实上这里没有一个很绝对的故障率标线，而且不存在绝对的 PT；只有某种程度上的 Robust。

Best Practice

CA, not P

我们放弃 Partition Tolerance，而尽力保证 Consistency 和 Availability：这是不可能的。在复杂而糟糕的现实世界中，Partition Tolerance 是不可避免的；消息丢失和系统故障永远出现。因此放弃 P 绝对是糟糕的主意。

AP, not C

我们放弃 Consistency（一致性），着力保证 Availability（可用性）。

我们的目的是保证用户体验，用户总能操作成功（或者说总能得到操作结果），即使数据可能不一致。

CP, not A

我们放弃 Availability（可用性），但是着力保证 Consistency（一致性）。

我们的目的是必须保证一致性，在此基础上没有办法地放弃 Availability；如当发生区域性断网时，拒绝新的用户请求直到网络恢复。类似于这次支付宝的 2 小时瘫痪，就是在 Partition 的情况下，为了保证 Consistency，无奈放弃 Availability 的结果。

Examples

选择 AP 还是 CP？取决于我们的目标要求。

类似于点外卖、在线购物这种无关紧要的内容，我们自然可以先暂时放弃 Consistency，到最后再去善后；

然而对于支付宝这种跟金融有关的重要事务，不能保证一致性的后果是非常严重的；所以不得不选择 CP。

Partially Choice

有时候，并非一定要从 AP 和 CP 中二选一；我们存在一个折衷方案可选。

例如，有两个区域欧洲和北美，网络断开后北美可以用，只有欧洲不可用；当网络恢复后，将北美的数据同步回欧洲，即北美为主，欧洲从属……之类的。

Transaction

Transaction，事务。

Facing Difficulties

我们目前的世界总是离理想太远。

我们所有的目标都是在一堆不可靠的结点上构造出一个可靠的系统。

极其困难的事情。

Basic Promises

T1

begin

transfer(A, B, 20)

withdraw(B, 10)

end

T2

begin

transfer(B, C, 5)

deposit(A, 5)

end

每一个「事务」都抽象为这么一个模型：BEGIN、BODY、END。

BODY 跟 END 都是套话——BODY 里面也就是每一个 Transaction 要干的内容。

然而，关键在于 T1 跟 T2 之间存在一个严格的保证：

- All-or-nothing
 - 每个 Transaction 中的 Body 指令，要么不执行，要么全部执行。不可以存在执行一半的情况。
- Before-or-after
 - T1、T2 必须（表面上）按顺序执行；即使是在多线程情况下也是如此。

这两项保证就足够实现我们的抽象系统了。

Atomicity

Atomicity: All-or-Nothing

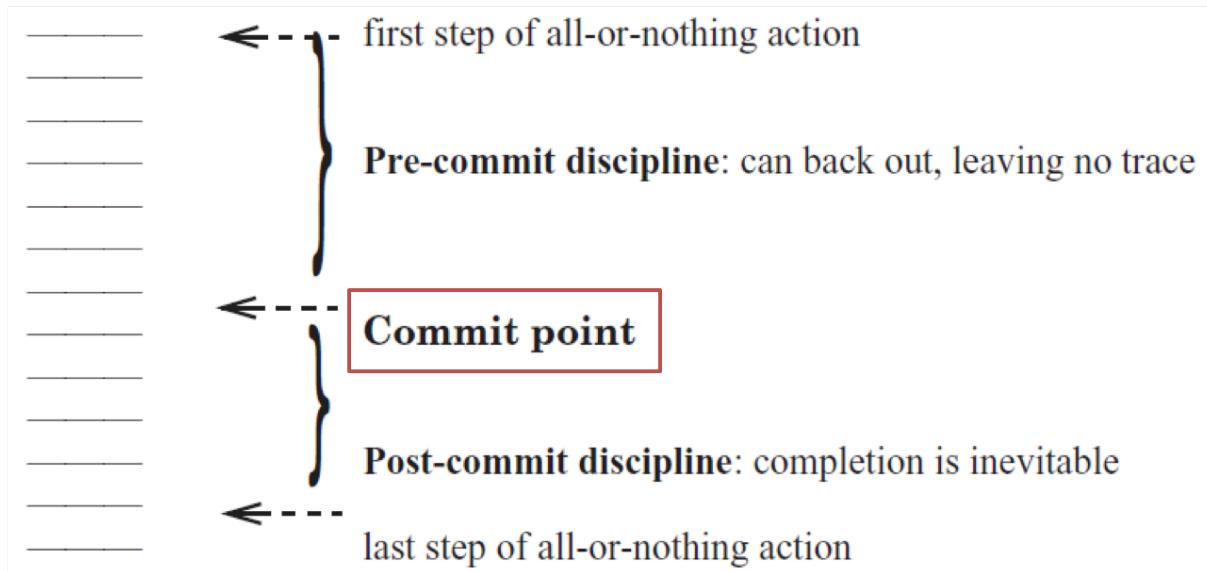
- An action is **atomic** if happens completely or not at all
 - If guarantee atomicity, it will be much easier to reason about failures
 - Need to think about the consequences of the action happening or not happening, but not about the action partially happening

All or Nothing 保证的是原子性（Atomicity）。对于每一个事务，我们要考虑的事情只有「他发生了吗？」、「事务之间发生的顺序是什么？」，而不考虑每个事务执行之间发生的事情，也就是认为每个事务是「原子的」、「不可分的」。

Commit Point

「提交点」，用图来看就是这样的：

Commit Point



15

对一个 Transaction 来说，在没有插入 Commit Point 之前，之前执行的 Pre-commit Discipline 都是可以撤销的，且不留痕迹（trace）。然而一旦插入了 Commit Point，那么还没执行的那些 Post-commit Discipline 都是不可撤销的（Inevitable）了，即使还没有进入，但是也非做不可了。

Commit Point

- Where is the commit point?

data state:	1	2	3	4	5	6	7
sector S1	old	bad	new	new	new	new	new
sector S2	old	old	old	bad	new	new	new
sector S3	old	old	old	old	old	bad	new

在我们按照顺序更新数据的过程中途中断的情况下，直接按照 Sector 的多数决就可以了。在数据状态达到 Data State 5 的时候实际上就已经算是绝对完成了。

然而 Data State 4 是不确定的：这种时候我们强行规定它 Backout。只是个规定。

Shadow Copy

影拷贝。何谓影拷贝？

我们以一个银行系统作为例子。

```
audit(bank):  
xfer(bank, a, b, amt):  
    bank[a] = bank[a] - amt  
    bank[b] = bank[b] + amt  
  
audit(bank):  
    sum = 0  
    for acct in bank:  
        sum = sum + bank[acct]  
    return sum
```

我们有两个可以调用的程序：

一个是 xfer(bank, a, b, amt) 可以在指定银行中新建一笔从 a 到 b，价值 amt 的转账；

另一个是 audit(bank) 用于统计指定银行中的所有款额。

xfer

如果我们按照上面的做法来实现 xfer 函数，必定无法实现原子性的保证。

这里我们采用了一个影拷贝的形式；

Shadow Copy

xfer(bank, a, b, amt):

```
bank[a] = read_accounts(bankfile)
bank[a] = bank[a] - amt
bank[b] = bank[b] + amt
write_accounts(#bankfile)
rename("#bankfile", bankfile)
```

在我们每次执行 xfer 的时候，先隐含地把银行数据 Copy 一份，并以 # 前缀将其临时保存下来；最后，再通过一个重命名将其放回原处。

Rename Procedure

对于 rename("#bf", "bf") 具体的 rename 过程是这几步：

- 1. 将 bf 对应的 inode 改写为 #bf 对应的 inode
- 2. 增加 Shadow Block 对应的 refcount
- 3. 减少正牌 Block 原来的 refcount
- 4. 删除 Shadow Block，将其对应的 refcount 减少

Crash Issues

问题来了：假如在这一过程之中发生了 Crash，会不会产生问题？

想象一下在我们上面的 RENAME 过程里，在第一步和第二步之间发生 Crash，导致两个 Block 指向同样的 inode（经过了 Step 1），然而其 refcount 没有递增（没有走到 Step 2），怎么办？这样两个 Reference 具有相同地位，无法判定谁是正身？

Fix: Rename Procedure

| 那么我们先更动 refcount，再改写，如何？

- 1. 增加 Shadow Block 对应的 refcount
- 2. 将 bf 对应的 inode 改写为 #bf 对应的 inode`
- 3. 减少正牌 Block 原来的 refcount
- 4. 删除 Shadow Block，将其对应的 refcount 减少

这样最坏的情况也不过就是造成 refcount 虚高，inode 释放受阻；但总归不会有大问题；而且这个经过一次 Scan 就能解决。

Summary

Shadow Copy

- Write to a copy of data, atomically switch to new copy
- Switching can be done with one all-or-nothing operation (sector write)
- Requires a small amount of all-or-nothing atomicity from lower layer (disk)
- Main rule: **only make one write to current/live copy of data**
 - In our example, sector write for rename
 - Creates a well-defined commit point

那么这就是我们的 Shadow Copy Rules 了。

Logging for AoN

还是以我们的银行系统里的 xfer 实现为例。

借助 Transaction 的保证，我们就可以这么实现：

```
xfer(bank, a, b, amt):
```

```
begin
```

```
    write (a, read(a) - amt)
```

```
    write (b, read(b) + amt)
```

```
    if read (a) < 0:
```

```
        abort //Not enough funds
```

```
    else:
```

```
        commit
```

以这个做法为例，它所打出的 Logging 大概是这样：

TID	T1	T1	T1	T2	T2	T2	T3
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50	COMMIT	A=80	B=70	COMMIT	A=110

```
begin // T1
A = 100
B = 50
"commit // A=100; B=50

begin // T2
A = A - 20
B = B + 20
"commit // A=80; B=70

begin // T3
A = A + 30 // A=110
--CRASH--
```

总之，下面五个事件是跟事务有关系的：

- Begin
- Write variable
- Read variable
- Commit
- Abort

Begin

Begin: allocates a new transaction ID

Write

Write: appends an entry to the log

Read

Read: scans the log looking for last committed value

Commit

Commit: writes a commit record

Abort

Abort: writes an abort record / simply does nothing

Recover

Recover: needless to do anything

简单说，将更动原来的数据给分离称在分开的 Transaction 里打 Loggings，在 Commit 的时候完全写入。

如果需要 Abort，可以直接放弃掉这块 Transaction，也可以写一个 ABORT 命令（不强制）。

Logging Issues

上面我们的 Logging Approach 看起来还不错，写入起来也还挺好的。

非常非常遗憾的是，Read 是非常差劲的。因为上面看到，每次我们要进行一个 Read，就非得 Scan 整个 Logging 不可。

因此我们使用一个 Replica 的做法来实现：对每一个 Transaction，既保留一份上面的 Logging，同时还保存一份 Cell Storage（格化的存储），可以快速地做 Read 而不必扫描全部的 Logging。

只不过要额外留意一下：在出现 Crash 的时候，我们应该把 Cell Storage 给清空，通过重新扫描 Log 来实现 Read。

Optimization

Cache Line

我们还可以直接将我们的 Cell Storage 给改成一个基于 LRU 或 LFU 的 Cache Line。

这样不至于让每次 Write 都需要执行两份工作量（写入 Log 和 Cell），而是分摊到 Cache Missed Read 中更好一点。

Log Truncate

目前的实现，Logging 的长度是完全不设限的；很可能会无限扩张。

我们应该在 Logging 超过一定长度的时候，选择将其割断（Truncate），在某些特定的点（叫做 Checkpoint）处，将比这更早的 Operation 都压平，以限制 Logging 大小。

Synchronize with I/O

Doesn't matter here...

Before or After

这里，我们开始考虑如何真正地实现 Before or After；即，保证表面上的「一先一后」。在多核心多线程的环境下，这件事并不那么容易。

Race Condition

最容易出现的问题就是 Race Condition 了：在两个线程交错执行的情况下，几乎没法保证最终结果的正确性。

Thread 1	Thread 2	Integer value
		0
read value	←	0
increase value		0
write back	→	1
	read value	← 1
	increase value	1
	write back	→ 2

Thread 1	Thread 2	Integer value
		0
read value	←	0
	read value	← 0
increase value		0
	increase value	0
write back	→	1
	write back	→ 1

当然我们可以严厉地要求两个线程必须工作在 SERIALLY 模式下（严格按顺序）；但那样就无法发挥多线程多核心的优势了。

Compromise Solution

我们最终的目的还是在「不可靠」之上抽象出「可靠」。尽量不要给底层强加限制，那样没法保障效率。

回忆我们在 All or Nothing 中所做的事情：我们引入了 Transaction 抽象，借此保证了 AoN。

而在 BoA 之中，我们也要做类似的事情：这里我们的抽象是这两位：

- 2PL (Two-phase locking)
- OCC (Optimistic Concurrency Control)

待会我们详细讲讲。

Goal

最后，我们提一句我们的最终目标：目标是什么？

run transactions T1, T2, .., TN concurrently, and have it "appeared" as if they ran sequentially

本质上让一系列 Transaction 并行执行，但却让他「表现得」像是顺序执行的一样。

...appeared?

表面上看起来是什么意思？

意思就是，我们可以随便让 T1 跟 T2 的 Body 执行，但我们要保证最终产生的结果要么是 SEQ(T1, T2) 的结果，要么是 SEQ(T2, T1) 的结果。

A schedule is final-state serializable if its final written state is equivalent to that of some serial schedule.

只要最终的结果是排列组合原子的 Transaction 所能拿到的结果，那么就认定为一个合理的调度 (Schedule)。

Conflict Serializability

有冲突的顺序化。

在什么情况下会出现两个操作冲突的情况？

但或许首先该问：啥是冲突？

Conflict

这里有一些规则：

- **Two operations conflict if :**
 1. they operate on the same object, and
 2. at least one of them is write
 3. the actions belong to different transactions
- **Conflict serializability**
 - A schedule is **conflict serializable** if [the order of its conflicts](#) (the order in which the conflicting operations occur) is the same as [the order of conflicts in some sequential schedule](#)

1. 两个操作操作了同一个对象
2. 其中至少有一个是 WRITE
3. 这两个操作分属不同的 Transaction

Serializability

如果一个 Schedule 中所有的冲突出现的顺序都和一个原子性的正常 Schedule 中的冲突先后顺序一致，那么就把这个叫做 Conflict Serializability。

这个判别法看起来好简单，实际上无法实际使用的。

后面会提到我们该用什么。

Conflict Graph

我们把每个 Transaction 画成 Graph 中的结点（Node）；然后，把有冲突的两个结点连接在一起。

留意这个边是有向的；由冲突始发结点指向冲突收束结点。

结论：如果这个 Graph 是 acyclic (非周期性の、非環式の) 的话，这一系列 Transaction 就是 C.S. (可冲突序列化) 的了。

Conflict Equal

如果两个 Transaction Schedule 的 Conflicts Order 一致，我们就称他们二位是 Conflict Equal 的。

View Serializability

A schedule is view serializable if the final written state as well as intermediate reads are the same as in some serial schedule.

假如一个 Schedule 最终的结果以及中间包含的读结果也都跟某个特定的 Serial Schedule 的结果相同，那么可以说这个 Schedule 是 View Serializable 的。

简单说，Final-state Serializable 只关心最终对数据产生的影响，而不管程序内部的 READ 是不是正确；

View Serializable 关心了读和最终结果。

Conflict Serializable 最严苛；他要求 Data Dependency 都不出错。

Final-state Serializable

Care the final state only

View Serializable

Care the final state as well as intermediate read

Conflict Serializable

Care the final state as well as all the **data dependency**

但是生成 View S. 比起生成 Conflict S. 要更困难一些：判断图是否成环还是不难的；但是要判断 VS 就很难了，是个 NP-Hard 问题。

况且，一个 Conflict S. 的调度一定也是 View S. 的（这显然），所以我们在实践中一般追求一个 Conflict Serializability 的调度。

Generate Conflict Serializable Schedules

最后一个问题：怎么生成 Conflict Serializable 的 Schedules 呢？

Pessimistic

悲观的人对什么都不信任，所以她选择给任何有可能产生冲突的地方挂锁，以此来防止产生数据的 Race Condition。

基于这种思路的策略包括：Global Lock, 2-Phase Locking。

Optimistic

乐观的人总是倾向于觉得所有的事情都不会出问题。

所以他们什么都不加限制，直到出现了问题，说：Conflict! Abort, clear history and retry!

基于这种思路的策略主要是 OCC。

Global Lock

这个锁就非常暴力了…全局挂一把大锁，每进入一个 Transaction 就拿锁，退出 Transaction 就解锁。

| 那这怎么可能会不 Serializable……你这个本身就是 Serialized

Simple Lock

稍微正常一点的上锁办法。

这个上锁办法的关键在于：

- 在进入 Transaction 之前，预先把所有会用到的 Shared Data 都上锁
- 遇到了 Commit 或者 Abort 之后就按照拿锁的反序放锁

这个有一点好处：完全没有利益相关的 Transaction 就不会 Block 彼此了。

但是存在问题：

- 怎么 Enumerate (遍历) 全部需要加锁的 Shared Object ?
- 或许我们加锁的对象可能比实际访问的对象要更多。这就太浪费了.....

Two-phase Locking

两步加锁法。听名字就很精妙（

- 每一个共享变量天生就带着一把自己的锁
- 要对这个变量进行任何操作，都必须先拿跟这个变量对应的锁
- 一旦一个 Transaction 释放了一把锁之后，他就不能再拿新锁了

| ? ? ?

这种加锁的方法，好处主要在不需要提前知道到底需要拿锁集合。而是采用「到了要访问数据的最后关头，再去拿锁」这种策略。

这样能使拿锁时间段最短，最小地影响其他 Transactions。

Fix: Two-phase Locking

两步拿锁法有个致命的问题：可能会死锁。我等你，你等我。

因为没有办法保证顺序拿锁反序放锁。

所以有两个解决方案：

1. 提供一个全局的拿锁顺序 (Dirty，不推荐)
2. 出现死锁，就 Abort 其中一个 Transaction 再来！

方法 2 也是基于上面的 Optimistic 的解决方案，而且很好地利用了原子性。绝好。

Concurrency Control

| 并行控制策略

CC 就是 Concurrency Control，也就是并行控制策略，负责调度那些运行时间上有重合的进程们。与此同时，还要保证上面提到的 Serializability。

这对于一个 CC 来说，并不是一件容易做的事情。

因此，Kung 和 Johnson 提出了一套基于乐观心态的 OCC 策略。

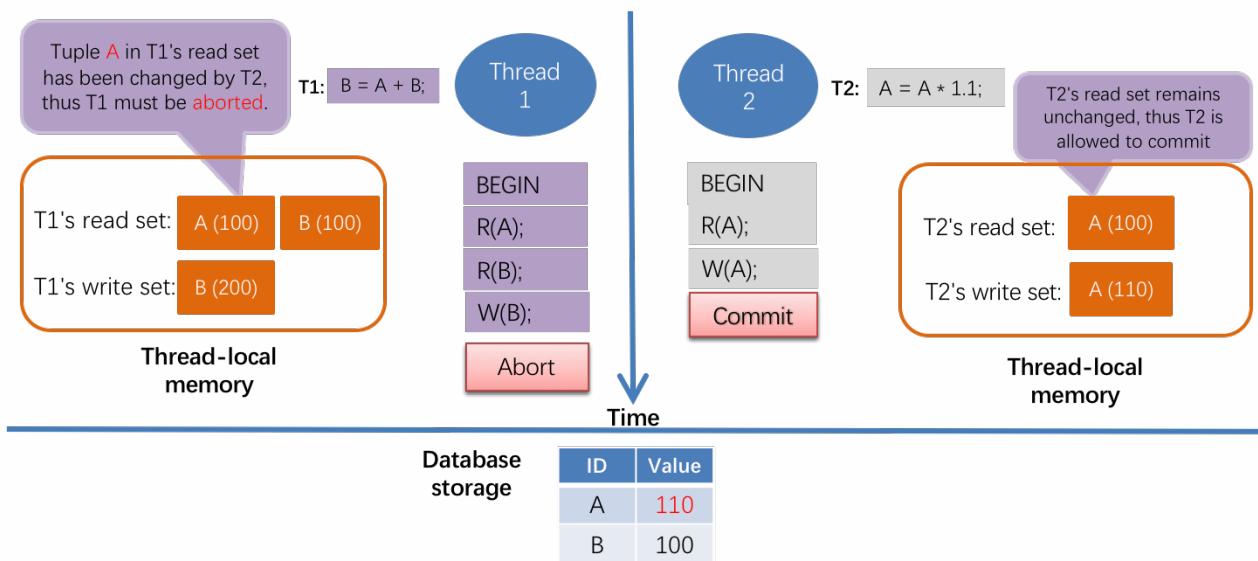
OCC: Optimistic CC

乐观的并行控制策略。

OCC Executes a Transaction in Three Phases

- **Phase 1: Concurrent local processing**
 - Reads tuples into a read set
 - Buffers writes into a write set
- **Phase 2: Validation in critical section**
 - Validates whether serializability is guaranteed:
 - Has any tuple in the read set been modified?
- **Phase 3: Commit the results in critical section or abort**
 - Aborts: aborts the transaction if validation fails
 - Commits: installs the write set and commits the transaction

OCC: An Example



Lecture 16: Lock & Memory Model

锁与内存模型

锁，实在是太重要了。给出了极致简单的抽象和严格的正确性保证。

而且，它为我们在不可靠的基础上抽象出一层可靠的系统提供了保证。

Implementation

那么我们该怎么实现锁呢？

Naïve (and Incorrect) Implementation

```
struct lock { integer state; };

void acquire (lock reference L) {
    while L.state == LOCKED
        ;
    // spin until L is UNLOCKED
    L.state = LOCKED; // the while test failed, got the lock
}

void release (lock reference L) {
    L.state = UNLOCKED;
}
```

或许所有希望制造锁的人最开始都是这么想的。如果拿锁被拒就循环着等待，退出循环就代表拿到了。

真的吗？

不。鉴于 acquire 锁所做的步骤包括了下面两步：

1. 读取锁 L 的状态
2. 将锁 L 的状态设为 LOCKED

这套序列不是原子的。也就是说，在 1. 和 2. 之间可能插入别的事件。或许一个线程插入了 1. 2. 之间，错误地认为它可以拿到锁，结果导致锁被多人持有。这显然是不对的。

Primitive Implementation

早期智人尝试过两种不同的锁的实现方法：纯软件实现和掺杂硬件的实现。

软件实现主要有：

- Using Load and Store instructions only
- Dekker's & Peterson's Algorithms

硬件实现主要有：

- RSM instruction: Read and Set Memory
- T & S: Test and Set
- C & S: Compare and Swap
- Load-linked + Store-conditional
- F & A: Fetch and Add

Peterson's Algorithms

```
int flag[2]; // assume two threads on two CPUs
int turn;
void init() {
    flag[0] = flag[1] = 0; // 1->thread wants to grab lock
    turn = 0; // whose turn? (thread 0 or 1?)
}
void lock() {
    flag[self] = 1; // self: thread ID of caller
    turn = 1 - self; // make it other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait
}
void unlock() {
    flag[self] = 0; // simply undo your intent
}
```

分析一下，可以看出来这段代码做的事情在于给每一个 Thread 都提供一个 Flag 位。同时，全局保留一个变量 turn 来记录当前拿锁的线程号。

在请求拿锁的时候，先把自己试图拿锁这一信息设定为 true (`flag[self] = 1;`)，并将 turn 设置为对方。

如果 `turn = 1 - self` 成立，且对方仍然持有这把锁，即这仍然是对方的回合，那么就持续 Spin。

这种办法依赖于一个假设，即所有的 LOAD 和 STORE 都满足原子性，即依赖内存系统的正确性。

在上古时代这是对的，然而现在的内存模型已经发展到不满足这件事了（笑）。

所以大家也不用这种锁了。

等等……插一句，讲讲内存模型是怎么一回事。

Memory Model

内存模型。

Definition

确定了 CPU 跟内存如何进行交互的模型。

这很重要，不同的模型可能完全改变 CPU 执行的结果。

Memory Consistency

对于内存一致性最朴素的理解就是：对同一块内存空间来说，Read 总是读到最后一次 Write 的内容。

Multi-Core

现在的 CPU 几乎都有多核心了。不同的核心却是共享着同一片主存的。他们会在完全随机的时间进行许许多多的读和写。

这件事可不简单。

Consistency Strategy

有哪些保持一致性的策略呢？

Strict Consistency

严格一致模式。

Strict Consistency

P1: W(x)1

P2: R(x)1 R(x)1

- Processor P1 writes a value of 1 to variable x; at some later time processor P2 reads x and obtains a value of 1. Then P2 reads it again and should get the same value

P1: W(x)1

P2: R(x)0 R(x)1

P1: W(x)1

P2: R(x)0 R(x)1

This is **not** valid under strict consistency

举例来说，两个处理器核心 P1 和 P2 同时工作，但是 Strict Consistency 的要求完全没有减弱；它要求在任何情况下，任何的读都应该读出任意最近的写的数据，无论他们来自哪一核心。

这个要求显然是很严格的（没法更严格了），基本上是把单核心的策略照搬到多核心来了。

可想而知，要保证这种一致性的代价很高。

Sequential Consistency

顺序一致模式。

孔子有云，

...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

翻译一下就是，只要访存的结果跟所有处理器严格按照顺序执行的结果一样（而不是瞎混在一起的），那就把它认定为一个顺序一致的访存。

```

P1: W(x)1
-----
P2: R(x)1 R(x)2
-----
P3: R(x)1 R(x)2
-----
P4: W(x)2
    
    Equivalent
P1: W(x)1
-----
P2: R(x)1 R(x)2
-----
P3: R(x)1 R(x)2
-----
P4: W(x)2

```

Sequential Consistency

Any ordering that could have been produced by a strict ordering regardless of processor speeds is valid under sequential consistency

We can reason about the program itself, with less interference from the details of the hardware on which it is running

```

P1: W(x)1
-----
P2: R(x)1 R(x)2
-----
P3: R(x)2 R(x)1
-----
P4: W(x)2

```

This is **not** valid under sequential consistency.

举例如图一中 P1 和 P4 写入了 1 和 2；而 P2 和 P3 则是连续进行了两次读取。

这样，P2（或 P3，他们是等地位的）可能读取出 1、1，也就是执行顺序为 P4 -> P1 -> P2_1 -> P2_2；

也可能读取出 2、2，执行顺序是 P1 -> P4 -> P2_1 -> P2_2；也可能读取出 1、2。

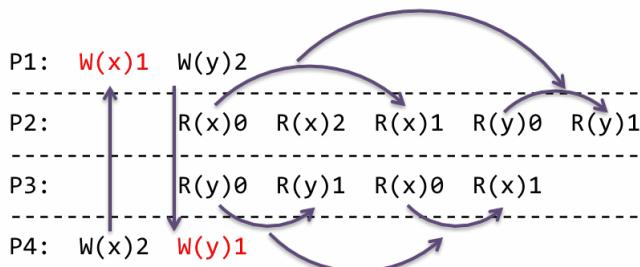
但如果读出 2、1，那就是违背了 Sequential Consistency 了。

因为不存在一个处理器的交替执行顺序能满足 2、1 的。

| Sequential Consistency 比 Strict Consistency 要弱一些。这个大家都看得出吧。

Cache Coherence

Cache Coherence != Sequential Consistency



Constrains:

- R(x): 0 -> 2 -> 1
- R(y): 0 -> 2 -> 1

P2 & P3 both think:

- P4's W(x)2 -> P1's W(x)1
- P1's W(y)2 -> P4's W(y)1

P2 thinks:

- P1's W(x)1 -> P4's W(y)1
- P3 thinks:
- P4's W(y)1 -> P1's W(x)1

P2 saw (x,y)=(1,0)

- P3 can never see it
- P3 saw (x,y)=(0,1)
- P2 can never see it

比较一下 Cache Coherence 和 Sequential Consistency。

不同之处是

- Sequential consistency requires a globally consistent view of memory operations
- Cache coherence only requires a locally consistent view
- e.g., for local variable x or y , all CPUs see the same order; but for (x,y) , different CPUs may see different orders

Seq Consistency 要求的是全局的一致性；然而 Cache Coherence 仅仅要求局部的一致性。

Process Consistency

| 处理器一致性（这啥？）

也称为 PRAM Consistency (Pipelined RAM Consistency)

定义是：

- Writes done by a single processor are received by all other processors in the issue order
- But writes from different processors may be seen in a different order by different processors

单个处理器所有的写操作都保证会被所有的处理器以相同的顺序接收。

但是，多个处理器的写操作可能会被不同的处理器以不同的顺序接收。

Processor Consistency

P1: $W(x)1$ $W(x)2$

P2: $R(x)2$ $R(x)1$

P1: $W(x)1$

This is **not** valid under processor consistency

P2: $R(x,y)1,0$ $R(x,y)1,1$

This is **not** valid under sequential consistency
But is valid under processor consistency

P3: $R(x,y)0,1$ $R(x,y)1,1$

- The processors are connected in a linear array, like this: 
- Cycle-1: P1 and P4 write their values and propagate them
- Cycle-2: the value from P1 has reached P2, the value from P4 has reached P3
- Cycle-3: the values have made it two hops

每个处理器接收「写」的顺序不同就导致了他可能「读」出不同的序列来。

Java Memory Model

JVM 是一层虚拟机，所以他也有自己的 Memory Model (哭)

它的宗旨是：

- 让用户做决定。默认情况下，只保证关键部分的 Sequential Consistency。

- 对非同步的多线程程序，只做最低程度的保证：保证你的读写操作一定发生在内存被初始化之後。
(这已经是……不怎么算保证的保证了)

Atomic Instructions

上面已经提到了，可以通过软件来实现锁。然后说来说去又提到了内存模型。

现在我们说回来：利用多功能原子指令来实现锁。

这种实现方法需要硬件提供特殊的指令集支持。

Test and Set

Test and Set 指令大概做什么呢？

用 C 代码来表示，大概是这样的：

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new; // store 'new' into old_ptr
    return old; // return the old value
}
```

将 `old_ptr` 的值设定为 `new`，并且返回设定之前的值。

注意，这整件事情是由指令集提供的原子指令。也就是说，你可以确保发生了一次从 `old` 改换成 `new` 的写入，而必不存在其他涉及到 `old_ptr` 的内存操作，无论写读。

这样，我们很快可以改写我们的 Lock 实现：

```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available, 1 that it is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

而且简洁，优雅，符合直觉（只有我做到了把 0 改成了 1，那才算是拿着锁了）。

Compare and Swap

这是 SPARC 的叫法。x86 叫做 Compare and Exchange。

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

基本操作是：只有 `ptr` 对应的内存是我希望得到的 `expected` 的时候，才把 `new` 写入内存。

本质上也是为了保证「我确实是把内存从 `expected` 改成了 `new`」。关键在于从 `expected`。

这就能保证不存在两个人同时进行 $0 \rightarrow 1$ 的改变，还都以为自己是那个起作用的人。

```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available, 1 that it is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (CompareAndSwap(&lock->flag, 0, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

同样简洁，优雅，符合直觉。

Load-linked and Store-conditional

| 这名字也太长

同样，上原子操作的等价 C 代码：

```

int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if /*no one has updated *ptr since the LoadLinked to this address*/) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}

```

这就不是很优美了...拆分成了两条语句，它保证的是在上次 Load 和这次 Store 之间如果没有别人更新过 ptr，就写入内存；否则拒绝写入。

这也就是上面的变种。关键在于实现 Read 跟 Write 的「无缝」链接，即其中没有插入别人的 Write。

```

1 void lock(lock_t *lock) {
2     while (1) {
3         while (LoadLinked(&lock->flag) == 1)
4             ; // spin until it's zero
5         if (StoreConditional(&lock->flag, 1) == 1)
6             return; // if set-it-to-1 was a success: all done
7             // otherwise: try it all over again
8     }
9 }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }

```

就是这代码没那么简洁明了了...你要给人解释 LL/SC 估计也得费点功夫。

Fetch and Add

老规矩，上 C：

```

int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}

```

这个就有点意思了。它解决了 $x += 1$ 无法保证原子性的问题：最经典的 Race Condition Demo 就是开两个线程分别给全局变量 $+= 1$ ，最后肯定加出来比正确结果少，是吧。

而如果用 FetchAndAdd 实现，那么就可以严格保证你的递增是正确的；不仅如此，你还能知道准确的递增之前（当然递增之后也就是 $+1$ 了）的值。绝对原子。

提供的这一功能所能实现的就是 Ticket Lock 了。

```
1 typedef struct __lock_t {  
2     int ticket;  
3     int turn;  
4 } lock_t;  
5  
6 void lock_init(lock_t *lock) {  
7     lock->ticket = 0;  
8     lock->turn = 0;  
9 }  
10  
11 void lock(lock_t *lock) {  
12     int myturn = FetchAndAdd(&lock->ticket);  
13     while (lock->turn != myturn)  
14         ; // spin  
15 }  
16  
17 void unlock(lock_t *lock) {  
18     lock->turn = lock->turn + 1;  
19 }
```



One difference with previous locks:
it ensures progress for all threads

基本策略是：FetchAndAdd 保证了每个请求锁的过程都会被分到独一无二的一个 Ticket，并且过程还能明确地知道这个 Ticket 的值。

如果仅仅有 Increase 而没有 Fetch，那么就无法保证拿 Ticket 的正确性了。

所以锁这个东西，一定是要把 Read 跟 Write 绑在一起，没有办法的。

Bootstrapping

| 引导，自力更生

- Solve the narrow problem using some specialized method
 - Might work for only that case
 - It takes advantage of the specific situation
- The general solution then consists of two parts:
 - a method for solving the special case
 - a method for reducing the general problem to the special case

如何解决一个很少出现的的 Corner Case ?

- 实现一个 Special Case 下可以用的解决方案，完全利用 Special Case 下 Special 的特征；

如何将其填补到通用的解决方案中去？

- 找出针对每一种 Special Case 的解决方案；
- 将通用的问题化归到特殊情况下去。

Lock Performance

锁的性能表现

首先，锁是一定会降低性能的。它的本质就是在合适的时候阻塞线程的运行，以保证某种程度上的正确性。

要求的正确性越严苛，阻塞的情况发生越多，那么锁的性能也就越差。

这是理所应当的。

Deadlock

性能差到极致，就是死锁。

可以简单理解成互相拿着对方要的东西，互相不撒手。结果双方都停止工作。

Deadlock's Rule

如果我们修改代码，保证每个线程拿到的锁都以反向释放，那么就可以保证一定不存在死锁问题。

这是可以被证明的。

反过来说，代码一点点微小改动都可能带来性能的大幅改善或恶化。

What causes Deadlock?

四个条件，构成充要：

- Limited access

资源有限，只能被有限个实例使用。

也就是说，存在有人需要排队的情况。

可以将互斥锁理解为「只有一个」的资源。

- No preemption

不存在相应的抢占策略：一旦拿到了资源，只有他自己有权释放。

- Multiple independent requests (hold and wait)

存在多个独立的请求：这会带来边 Hold 边 Wait 的情况。

换句话说，存在「一边拿着一些资源一边请求另外的资源」的情形。

- 在 wait-for 图中存在环。

这些看起来都是很自然的事情；但是要留意，这些是充分必要条件，也就是可以跟「有可能出现死锁现象」划等号的。

Avoiding

通常来说，只要在多线程里用到了锁，这件事情就是不可避免的。

我们通常有这么两类解决方案：一类积极的，一类消极的。

Pessimistic

消极方案是先做一个 *a priori*：沙盘推演。

先看看在目前的情况下我能拿到什么锁，不能拿到什么锁，是否有可能出现死锁。假如有可能出现，那我直接放弃所有拿锁。

Optimistic

积极方案是正常情况下不去考虑出现 Deadlock 的情况，而是等出现了之后再去考虑收拾他。

「在出现之後收拾」的办法，首先你得能 Detect 死锁的出现。

通常利用 Timeout 时间来判断。在检测出来之後，是 Abort 掉一个线程，还是用 Preemption 机制抢走某一个线程的锁，这都由你决定了。

Livelock

这个很奇怪：既然是 Live，就说明没有死锁，那有什么问题呢？

实际上，Live Lock 的定义是双方规律性地做同样的事情，并且在很特殊的情况下，这种不断的 Retry 刚好阻止了正常运转。

打个比方，双方同时伸手去摸一个文件，约定如果刚好碰到对方摸，那就等待 500 ms 之后重试。结果他们刚好每次都碰上手，每次双方都立刻缩回手，并且在 500 ms 后重复这一动作。结果大家都摸不到这个文件。

大概就是这么个意思。

Lecture 17: Thread & CV

| 此 CV 非彼 Computer Vision...

Intro

Producer & Consumer Problem

生产者 / 消费者问题。简单说，就是独立运行的两个线程进行互相影响的操作。

独立，是因为它们具有各自的逻辑和执行流程；互相影响，是基于他们共享的一块 Buffer（缓冲区）。

Producer 负责往缓冲区里放数据。假如缓冲区已经满了，它就必须等待。

Consumer 负责从缓冲区里拿出数据。假如缓冲区还是空的，它就必须等待。

用经典的流媒体播放器为例：后台的网络请求线程会不断地进行网络请求，往缓冲区里塞还没播放的影片；而前台的播放线程则会按照一定速度从缓冲区里拿数据，并实现连续播放的效果。

假如网络请求太快而播放线程跟不上，那么就会导致缓冲区塞满，网络请求器也不会继续进行请求。而假如网络请求太慢导致缓冲区时常净空，播放线程拿不到数据，则必须等待 Producer 的下一个生产动作。这时候就会「缓冲中...」了。

用 Xia Yubin 的话来讲，这种策略就是 enable communication while keep isolation，在保证隔离的情况下启用通信交流。

而那块 Buffer 就是交流的关键。

但在那些「大规模、分离式」的系统中，就没有那么容易用一块共享的 Buffer 作为 Shared 了；为了保证隔离化和模块化，一般我们会采用类似 RPC 这类技术来实现信息的交流。

在实际的计算机系统中，也有在 Kernel 态保留 Buffer、只允许通过特定 API 来读写 Buffer 的实例。

Bounded Buffer

Bounded Buffer & APIs

Bounded Buffer 就是我们上面提到的：空间有限的缓冲区。

他提供的 API 是这样的：

```
void send(void* content);
void* receive();
```

像一块甜蛋糕一样简单。但是留意到 `send` 和 `receive` 永远不会需要一个返回值指示操作能不能完成。

只要他们返回了，就代表成功完成了 Produce / Consume 的操作。换句话说，如果没能完成操作，他们就不会返回。

假如缓冲区满了 / 空的，那么他们根本不会返回控制流，而是会持续阻塞。

这种实现的问题就是长时间的 Hanging，吊着不返回。

况且，没有一个办法来实现 Sequence Coordination（顺序协调）。Consumer 的排队跟他们最终 Consume 的先后没有关系，完全是随机的。

Producer 也可以看作是争抢者；他们争抢 Buffer 的空位。

Bounded Buffer Implementation

```
send(bb, m):
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
    return
```



这是一种比较典型的 send 的实现。注意其中红色双向箭头指向的两行语句。是否可能交换他们？

当然不行了。在这里我们所做的是：先写入内存，再将 `bb.in + 1`。

但如果我们先写入 `bb.in`，再写内存 Buffer，那就意味着我们在实际做事之前先声张开来。

特别注意 Send 跟 Send 之间肯定是不能重合的，一定有锁保护。

| 事实上这个例子里只有一个 Sender 和一个 Receiver...

但 Send 跟 Receive 之间不存在互斥锁，他们有可能会交错。

所以我们一定要小心保证，不存在这种「错误的瞬态」。

下面是 Send 跟 Receive 的完整实现。

```
send(bb, m):
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
        return
```

```
receive(bb):
    while True:
        if bb.in > bb.out:
            m ← bb.buf[bb.out mod N]
            bb.out ← bb.out + 1
        return m
```

Implicit Assumptions

- 单个 Consumer，单个 Producer。
- 在各自的 CPU 上运行
- Bounded Buffer 的 in 和 out 变量不溢出
- Read 和 Write 操作都 Coherence
- Bounded Buffer 的 in 和 out 变量都满足 Before / After 的原子性
- The result of executing a statement becomes visible to other threads in program order

怎么理解这个呢？意思是说单个线程所有的执行语句在其他线程看起来都是同样顺序的（不会乱序）。

Optimizations

Concurrency

| 有办法把这个 bb 给拓展到多核处理器上吗 ?

事实上，这个 (Single P Single C) 的实现在双核 CPU 上工作完全良好。

Multiple Senders

| 不满足于 Single P Single C.....能给我多个 Sender 吗 ?

出现多个 Sender，就意味着他们可能同时调用 send 接口。

甚至在总线上，寻址指令可能都在同时。

很容易就能想出可能出问题的点：

A

bb.in = 0, bb.out = 0

B

bb.in = 0, bb.out = 0

write m1 to buf[0]

write m2 to buf[0]

set bb.in = 1

set bb.in = 1

m1 lost!

都以为自己作为 Producer 写入了一块内存，殊不知这块内存被两个人请求了。这样後写的人就白写了。

Locked Send

```
send(bb, message):
    while True:
        if bb.in - bb.out < N:
            acquire(bb.send_lock)
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
            release(bb.send_lock)
    return
```

这样加的锁，正确吗？

可能有两个 Sender 同时判断 `if bb.in - bb.out < N`，然后同时落入 TRUE 分支。

然而如果此时 Buffer 只有一个空位，那么就会造成在不该 Send 的时候发送 Send。

Locked Send: Corrected

```
send(bb, message):
    acquire(bb.send_lock)
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
            release(bb.send_lock)
    return
```

我们把判断 `< N` 的代码也放进锁来，就不是问题了。

Locked Send: Another

```

send(bb, message):
    while True:
        acquire(bb.send_lock)
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
        release(bb.send_lock)
    return
release(bb.send_lock)

```

只是换了一个包围锁的位置。基本类似。

yield()... What?

Intro

我们在使用「锁」的时候，会产生大量的 Spinning 的。也就是，不断地在各个线程之间切换，而其中大部分线程都是在 While 循环中 Spinning 着。

好浪费啊！有没有什么办法在条件不满足的时候，直接不要调度给他呢？

The Yield: System Call

这个 System Call 主要做这几件事情：

- Suspend Running Thread
 - 把调用这个 System Call 的线程挂起，但是记录下他的帧指针和页表指针
- Choose New Thread
 - 用 Round Robin 找下一个可以运行的线程，直到找到一个可以运行的线程（有可能就是他自己）
- Resume Thread to Run
 - 把那个线程叫醒，恢复上下文继续运行。

我们需要的数据结构是：

- threads table
- CPUs table
- t_lock

`send()` with `yield()`

```
send(bb, msg):
    acquire(bb.lock)
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- msg
            bb.in <- bb.in + 1
        release(bb.lock)
    return
release(bb.lock)
yield()
acquire(bb.lock)
```

用例子来讲一下。在我们发现 Send 条件不满足的时候，先把锁放了，然后 yield 自己等下次被唤醒。

这样，不会有失败的 Sender 始终拿着锁 Spin，而是智慧地放弃调度权等下次拿锁。

Yield's Implementation

```

yield():
    acquire(t_lock)

    id = cpus[CPU].thread
    threads[id].state = RUNNABLE
    threads[id].sp = SP

```

do:

```

        id = (id + 1) mod N
        while threads[id].state != RUNNABLE

```

```

        SP = threads[id].sp
        threads[id].state = RUNNING
        cpus[CPU].thread = id

```

```
release(t_lock)
```

YIELD() Implementation

} Suspend
running thread

} Choose new
thread

} Resume new
thread

t_lock

1. Atomically set threads[].state and .sp
2. Atomically find a RUNNABLE thread and mark it RUNNING

Yield 的具体实现是这个样子的。

假如所有的其他 thread 都不是 Runnable，那就会到她自己身上继续执行。

Conditional Variables

留意到上面的实现有个问题。有大量的 Check，而且反复地拿锁放锁也不好，很浪费。

有没有什么办法，让 Buffer 有空位的时候，才去唤醒 Sender Thread，而如果没有就让它继续睡？

天底下哪有这样的好事？

标题里的 CV 就是「条件变量」的意思。

Solution

- Condition Variable
 - Let threads wait for events
 - The threads get notified when the events occur
- Condition variable API
 - **WAIT(cv)**: yield processor and wait to be notified of cv
 - **NOTIFY(cv)**: notify waiting threads of cv

条件变量好处都有啥？

可以让 Thread 等待某些特定的事件发生。

APIs

- `wait(cv)` : 挂起调用这个函数的线程，等待 cv 把他唤醒
- `notify(cv)` : 把所有等待着 cv 的线程叫醒。当然没法自己把自己叫醒，因为自己挂着不可能调 Notify。

Send with WAIT/NOTIFY (Incorrect Version)

```

send(bb, msg):
    acquire(bb.lock)
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- msg
            bb.in <- bb.in + 1
            release(bb.lock)
            return
        release(bb.lock)
        yield()
        acquire(bb.lock)

send(bb, msg):
    acquire(bb.lock)
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- msg
            bb.in <- bb.in + 1
            release(bb.lock)
            |  notify(bb.not_empty)
            |  return
            release(bb.lock)
            |  wait(bb.not_full)    wait() will call yield()
            |  acquire(bb.lock)

```

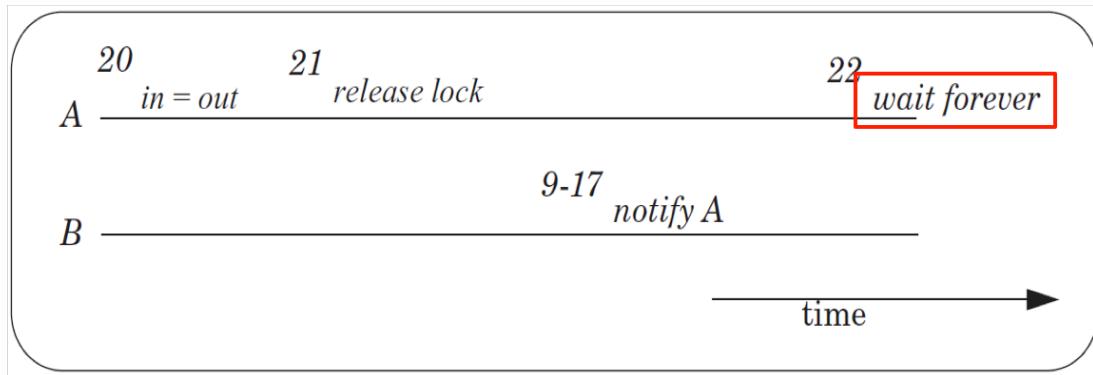
我们按照这个思路写了一个新的 Send。

然而这个是不对的。关键在于 Notify 可能会丢失。

详细解释一下，考虑这么一个情况。

放锁和 Notify 之间可能会插入其他语句，把 bb.lock 锁给拿走了。这样就导致 notify 失效。

The Lost Notify Problem



33

本质的问题在于：wait 和 notify 的微小先后次序会导致结果差异。

- Condition variable itself has no memory/state
 - wait() and then notify(): wait() returns
 - notify() and then wait(): wait() does not return
 - This is potentially prone to race conditions

先 wait 再 notify，wait 会被唤醒。

先 notify 再 wait，那就叫不醒了。

New APIs

- Old API
 - WAIT(cv): yield processor and wait to be notified of cv
 - NOTIFY(cv): notify waiting threads of cv
- New API
 - WAIT(cv, lock): release lock, yield CPU, wait to be notified
 - NOTIFY(cv): notify waiting threads of cv

既然之前是放锁和 wait 两件事之间产生了间隙导致的问题，那我们干脆直接把他们合成一个原子的好了。确保不会有人拿到了锁之后，在我 Wait 之前给我发个什么 Notify。那就倒霉了。

New API: WAIT(bb.full, bb.lock)

```

send(bb, msg):
    acquire(bb.lock)
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- msg
            bb.in <- bb.in + 1
            release(bb.lock)
            notify(bb.not_empty)
            return
        release(bb.lock)
        wait(bb.not_full)
        acquire(bb.lock)

send(bb, msg):
    acquire(bb.lock)
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- msg
            bb.in <- bb.in + 1
            release(bb.lock)
            notify(bb.not_empty)
            return
        release(bb.lock)
        wait(bb.not_full)
        acquire(bb.lock)
        wait(bb.not_full, bb.lock)

```

就可以改写成这样了。

Preemption

抢占。厉害！

Types

调度策略有下面几类：

- 非抢占调度

一个线程只要拿到了控制权，就有权一直运行下去，直到他自己放弃。

- 合作式调度

每个线程都认为彼此会周期性地调用 YIELD 来放弃控制权。

这样，他们就能放心地交出控制权，以确保自己将来还有机会。

- 抢占式调度

经过一定的时间，线程管理器强制让活跃线程交出控制权。

Preemptive

我们比较关心抢占式调度。本质上非抢占调度跟合作式调度基于对进程的信任。而抢占式是带有强制性的。

Implementation

最简单的实现是基于一个硬件时钟：每隔一段时间发一个中断。中断被截获之后，就可以在 Kernel Mode 里处理调度了。

中断处理程序也就是调度器呢…特别要注意的是他调用的内核态的 Yield 不是一个编译器生成的函数。

你需要自己处理寄存器的保存问题。自己把它们放到 Stack 里。

Summary

- Threads
 - Virtualize a processor so that we can share it among programs
 - yield() allows the kernel to suspend the current thread and resume another
- Condition Variables
 - Provide a more efficient API for threads
 - Threads wait for an event and are notified when it occurs
 - wait() requires a new version of yield(), yield_wait()
- Preemption
 - Forces a thread to be interrupted
 - So that do not have to rely on programmers correctly using yield()
 - Requires a special interrupt and hardware support to disable other interrupts
- 线程：处理器上的一层抽象。
- CV：提供了更高效的线程睡眠 / 唤醒 API。
- 抢占：强行夺取控制权。

Lecture 18: Distributed Transactions

Xia YB 的目标没有变：Build Reliable Systems from Unreliable Components。

Intro

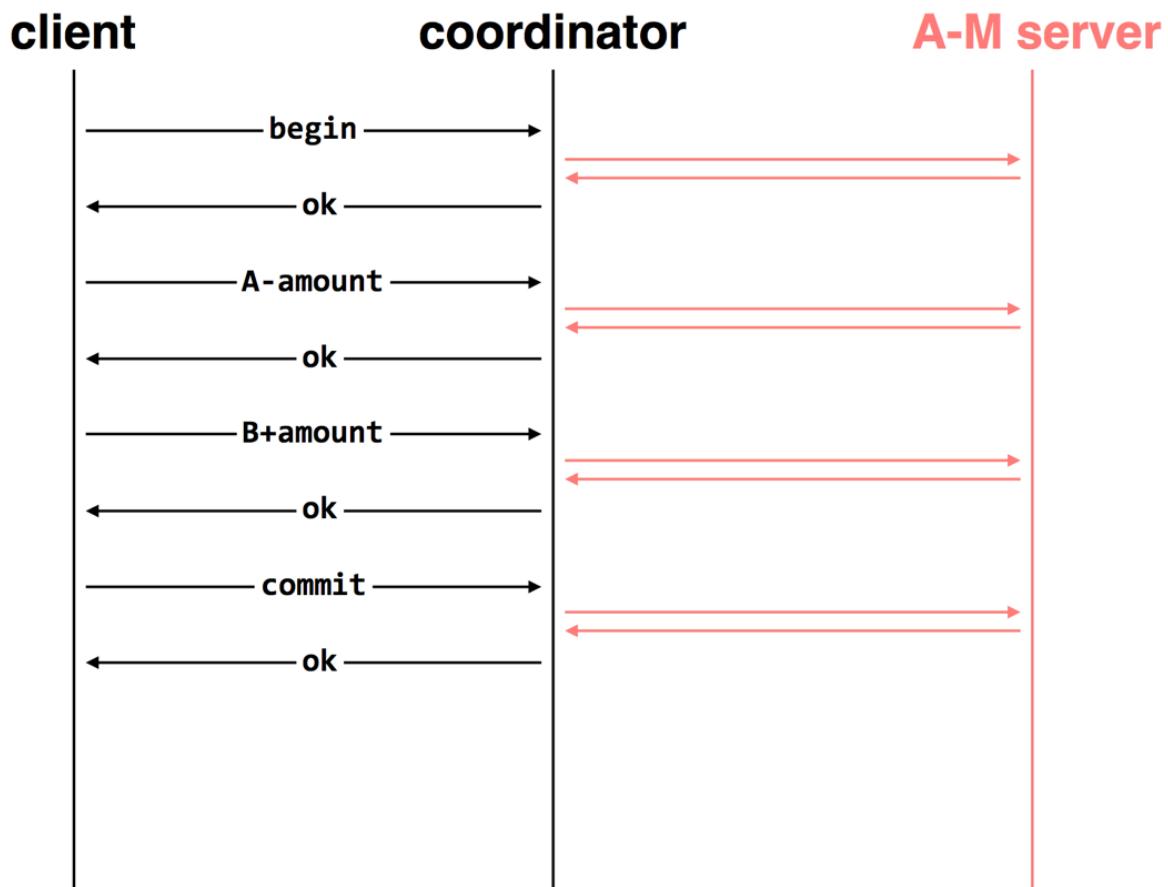
举个例子，我们的 Bank 系统大如磐石，必须分拆。我们规定由两台服务器提供服务，其中一台处理 A-M 的事务；另外一台处理 N-Z 的事务。

那么问题就来了：有可能有一些事务需要跨服务器进行。比如 Alice 要给 Zulu 转账，两台服务器都必须知晓并给出正确的记录。

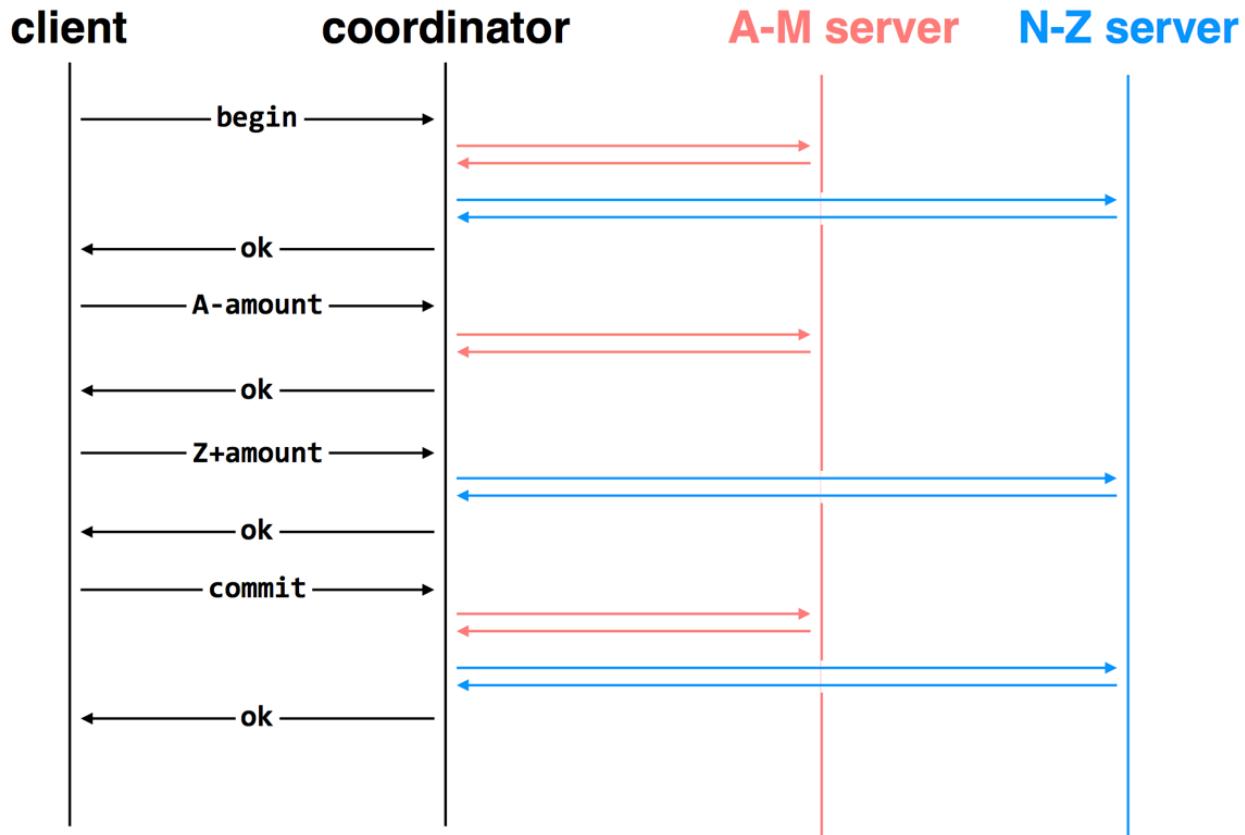
Coordinator

这就需要有一个协调人「Coordinator」负责服务器的调度。客户端不知道也不应当知道有几台服务器，自己该去找哪台服务器。客户端只需要找到 Coordinator，报上自己的大名，由协调人来帮助把这个请求转发给对应的服务器。

只涉及到一台服务器的请求很简单。



涉及到两台，因为有 Coordinator 的抽象，也还算能做。



Network

网络世界充满了延迟、复杂和丢失。

任何消息都可能会丢失、延迟到达或重复到达。

为了不在这一层上面浪费心思，我们选择用 RPC 来建构抽象。

- 通过 Persistent Sender 来保证「至少一次」
- 通过 Duplicate Suppression 来保证「至多一次」

非常遗憾，基于 RPC 的消息传送机制并非原子的。

我们在后面可以看到很多出错的情形。

Problems

多个服务器的情形很可能不一致。

比如，Coordinator 对两台服务器的 Commit，一个成功一个失败。甚至一个成功，一个直接 Crash。

那么对第一个「成功」的 Commit 也不算数了。但按照我们现在的做法，没法简单地把它给召回。（你还得考虑召回是不是成功，etc...麻烦大了）

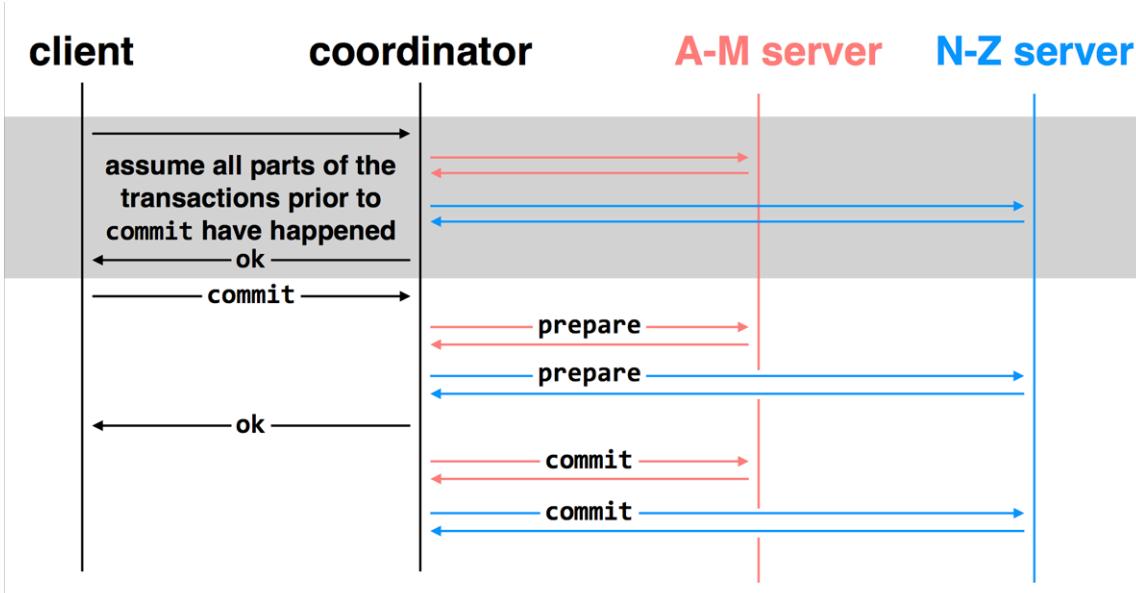
Solution: Two-phase Commit

两步 Commit 法。

把 Commit 分两步走。

- 第一步，叫做 Preparation 或者 Voting。
 - 这一步里，底层的 Transactions 要么 Abort (对应上面的 Commit 失败) ，要么 Tentatively committed (暂定成功，对应上面的 Commit 成功)。
 - 然后，顶层收集这些「一步 Commit」的信息：要么是 Abort 了，要么是「暂定 Commit」了。
- 第二步，才是真正的 Commitment。
 - 这一步里，只有顶层达成一致意见，也就是底层都做好了 Commit 的准备，才进行最后的 Commit 确认。
 - 否则，整个 Commit 大业尽毁。

Another Problems



- two-phase commit: nodes agree that they are ready to commit before committing

看起来很美好，大家都先做好准备，一致决定之後才去进行 Commit。

非常遗憾的是，消息传递是不可靠的。

甚至连「准备好了」 「没准备好」这种消息都没办法可靠地发送给网络中的其他结点。

我们必须提出新的概念来解决这个难题。

Multiple-Site Atomicity

多点式原子性。

Coordinator's Job

对于上面的协调人「Coordinator」来说，他的任务就是收集底层人民的呼声；

假如人民说 ABORT 或者不说话，那就说明不能 COMMIT，对外返回 ABORT。

| 更聪明一点的协调人会把任务分配给其他 Worker...以免脏节点污染

假如人民全部都说 COMMIT READY，那就对外发布 COMMIT 令。

Worker's Job

对于普罗大众来说，他们的任务就是：在什么都没收到的情况下，反复广播自己目前的状态 PREPARED，意为「我准备好进行 COMMIT 了！」。

而在收到 Coordinator 的 COMMIT 令後，直接调用 Commit 提交更改。

就是这么回事。

Worker's Failure

试想一下，假如一个 Worker 在 Commit Point 之后爆炸了，我们怎么办？

- 我们没办法 Abort 掉 Transaction。
- 我们希望 Worker 自己能从错误中恢复。
- 我们希望 Worker 能记录所有 Transaction 目前的状态（是不是 Prepared for Commit）。

因此，我们决定：

- Worker 一准备好，就把 Prepared 这件事写到自己的 Log 里。
- Worker 一旦 Crash 重启，会进入 Recovery Mode。
- 恢复模式下会读取 Log，并确定目前的准备状态。

Summary

- Two-phase commit allows us to achieve multi-site atomicity: transaction remains atomic even when they require communication with multiple machines
- In two-phase commit, failures prior to the commit point can be aborted. If workers (or the coordinator) fail after the commit point, they recover into the **PREPARED** state, and complete the transaction
- Our remaining issue deals with availability and replication: we will replicate data across sites to improve availability, but must deal with keeping multiple copies of the data consistent

Lecture 19: Distributed Transactions

| 接着讲分布式事务...

Replication Consistency

| 重复一致性

在多处重复存储的同一数据，怎么保证他们的一致性？

也包括积极和消极两种。

- 积极型

- 容忍（一段时间的）不一致，并在稍后修复他们。
- 假如「Out of Sync」（失去同步）的数据可以被容忍，那这种方法没问题。

- 消极型

- 一定要求备份之间具有强有力的一致性。
- 假如失去同步的数据会带来非常严重的问题，就用这种策略。

Optimistic Replication

积极型具体怎么实现的呢？

- 每次做修改的时候，都记录下进行操作的时间戳（Timestamp）。
- 在恢复同步之后，再根据修改时间决定保留谁。

| Strawman 曰：拥有最大 mtime 的文件才能被保留。

Clock Matters

要保证同步的结果符合预期，很重要的一件事就是保证「时钟」在各个机器中同步。

否则，生成的时间戳就不一致了。

因此，我们有必要对 Replica 中的机器进行 Clock Synchronizing（时钟同步），确保时间戳的计算基准一致。

Pessimistic Replication

消极的人做事会比较认真（暴论）

因此消极策略就必须做更多的事情来保证不出问题。

这种策略的代表是 RSM 和 Paxos。

Goal

我们最终的目标还是实现类似于单机的效果：As if there's only one single copy.

Strawman's Idea

- 客户端给每个服务器都发送一遍请求，但不要求他们都成功；有一台成功就算完。

遗憾的是可能出现 Network Partition：区隔问题。在这种情况下，每个客户端都只会连接一个固定的服务器，数据都只会进入那一区，Replica 就无从谈起。

Handling Network Partitions

客户端始终认定多数决。如果能访问到三台服务器，其中两台返回一致，那么两台就是多数决；如果只能访问两台服务器，或者三台服务器的数据都不一样，那么就只好等待了。

这种情况下，就能处理 Single Failure 的问题了。

Quorum

| 法定人数

你要求多少台服务器才能进行多数决？

总不能一台服务器你就「多数决」，那不成。

- Define separate read & write quorums: Q_r & Q_w
 - $Q_r + Q_w > N_{replicas}$ (Why?)
 - Confirm a write after writing to at least Q_w of replicas
 - Read at least Q_r agree on the data or witness value
- Example
 - In favor of reading: $N_{replicas} = 5$, $Q_w = 4$, $Q_r = 2$
 - In favor or updating: $N_{replicas} = 5$, $Q_w = 2$, $Q_r = 4$
 - Enhance availability by $Q_w = N_{replicas}$ & $Q_r = 1$

计算公式： $Q_r \& Q_w - Q_r + Q_w > N_{replicas}$ 。

只有此条件满足才足够。

Lecture 20: RSM & Paxos

还是接着我们上面的讲：消极的 Replica 策略。

RSM

全称 Replicated State Machines。

| 听着就专业。

Definition

- 通用的一套保证 Replica 一致性的方法。
 - 对于一组初始状态一致的服务器...
 - 对于按照相同顺序来到的相同 Input 的 Operation...
 - 在没有随机扰动的情况下...
 - 总会到达相同的终态。

这一策略提供的保证还是很强的。

Usage

在我们使用 RSM 的过程中，最难保证的就是：

如何确认对每一台服务器，所有请求的顺序都是一致的？

如果发给单台服务器的请求顺序不一致，那么 RSM 条件不满足，就可能结束于不同的终态。

P/B Model

Primary & Backup 模型。

也就是主 / 备模型。不可以让所有的服务器具有相同的地位，无法保证他们拿到的 Operation 的顺序一致。那就没法决定听谁的了。

General Idea

基本的想法是：Coordinator 只去跟 Primary Server 沟通交流，而由 Primary 服务器去跟众多 Backup 服务器沟通。

假如 Primary 服务器挂了，Coordinator 就会自动切换到 Backup 服务器提供服务，将他的地位提升到 Primary。

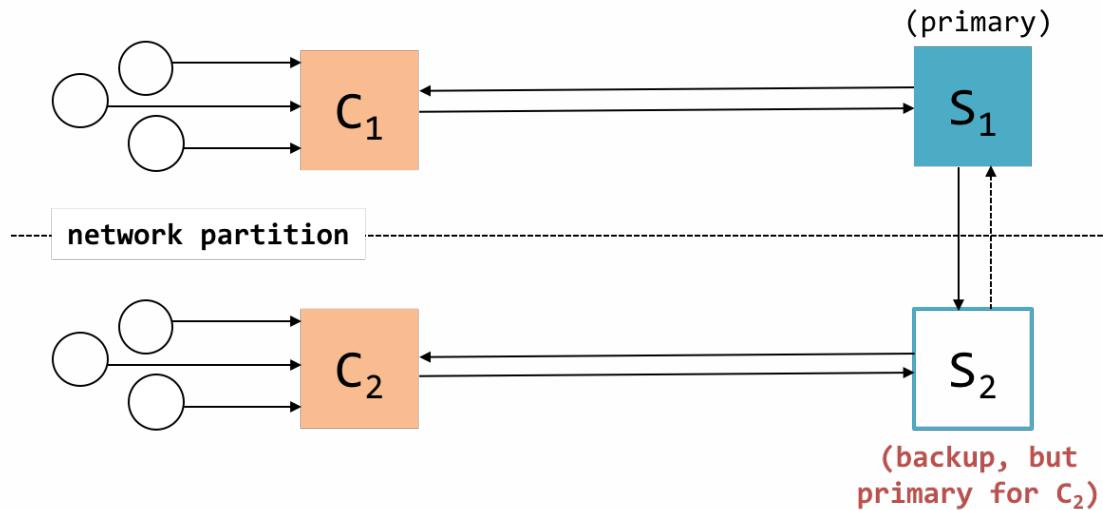
Network Partition Issues

在这种情况下遇到 NP 的时候，怎么办呢？

这时候可能不同的 Coordinator 会连接到不同的 Server，并且他们都认为自己正在连接的就是 Primary。

而不同的 Server 之间就存在不一致性，这问题就大了。

Multiple Coordinators + the Network = Problems



C_1 and C_2 are using different primaries;
 S_1 and S_2 are no longer **consistent**

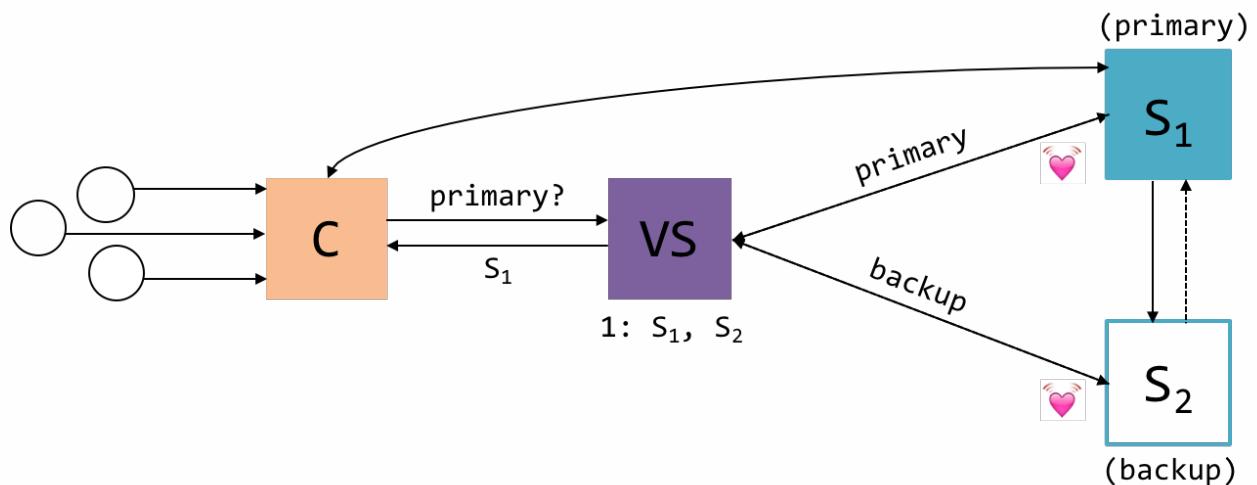
View Server

视图服务器

看起来仅凭 Coordinator 跟 Server 还是会犯错，他们都看不到全局。

需要一个在高点观察整个系统的服务器来解决问题。

经由 View Server 来决定谁是 Primary，谁们是 Backup。



现在结构图看起来是这样。

VS 何德何能就能处理好 Network Partition 的问题呢？

因为它遵循下面一些规则：

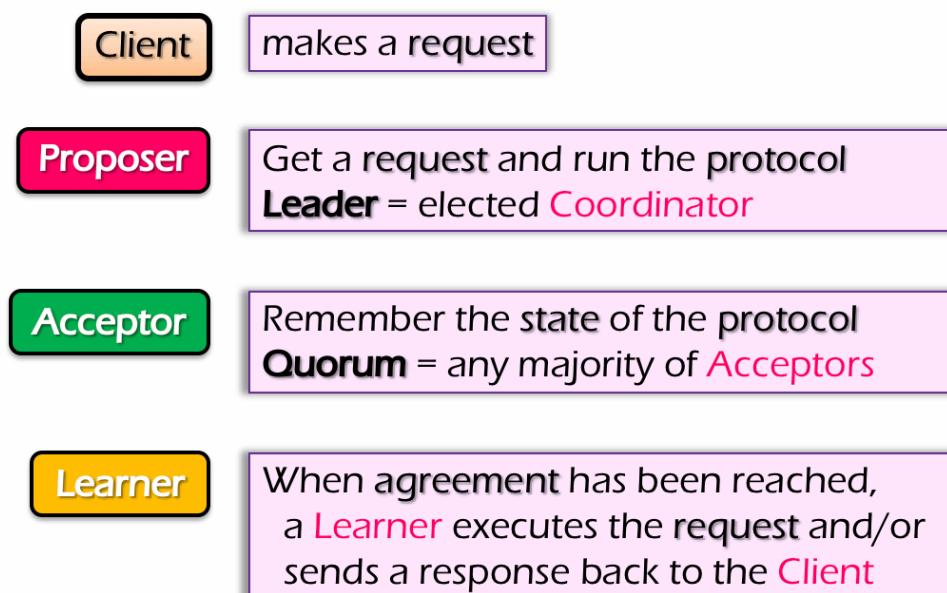
1. Primary must wait for backup to accept each request

2. Non-primary must reject direct coordinator requests
 - That's what happened in the earlier failure, in the interim between the failure and S2 hearing that it was primary
3. Primary must reject forwarded requests
 - i.e., it won't accept an update from the backup
4. Primary in view i must have been primary or backup in view $i-1$

Paxos

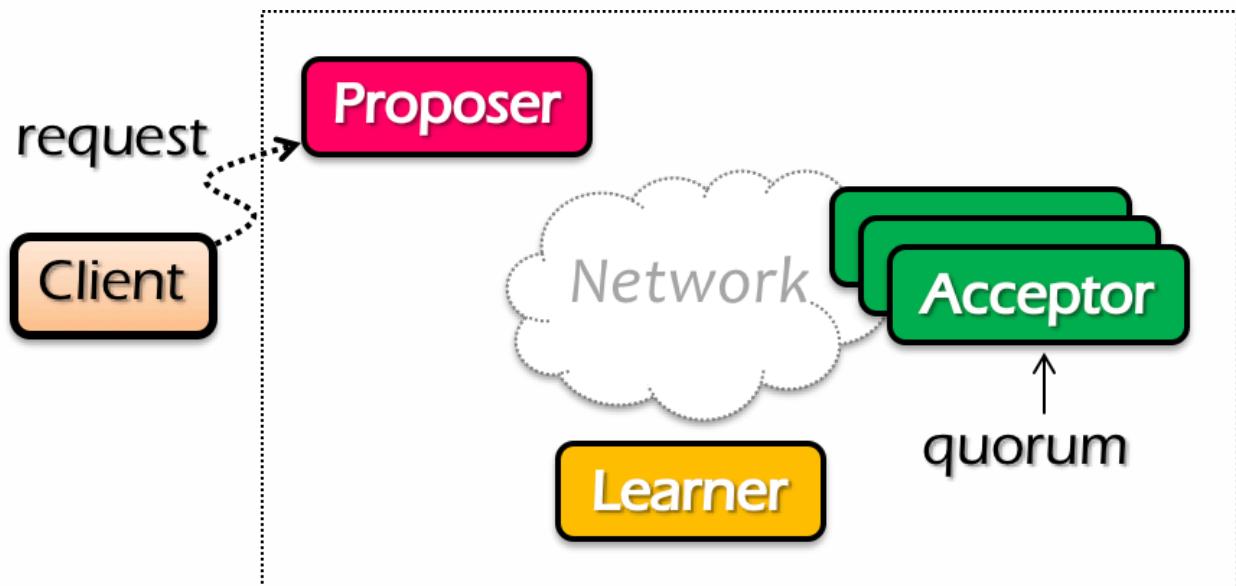
Structures

Paxos Players

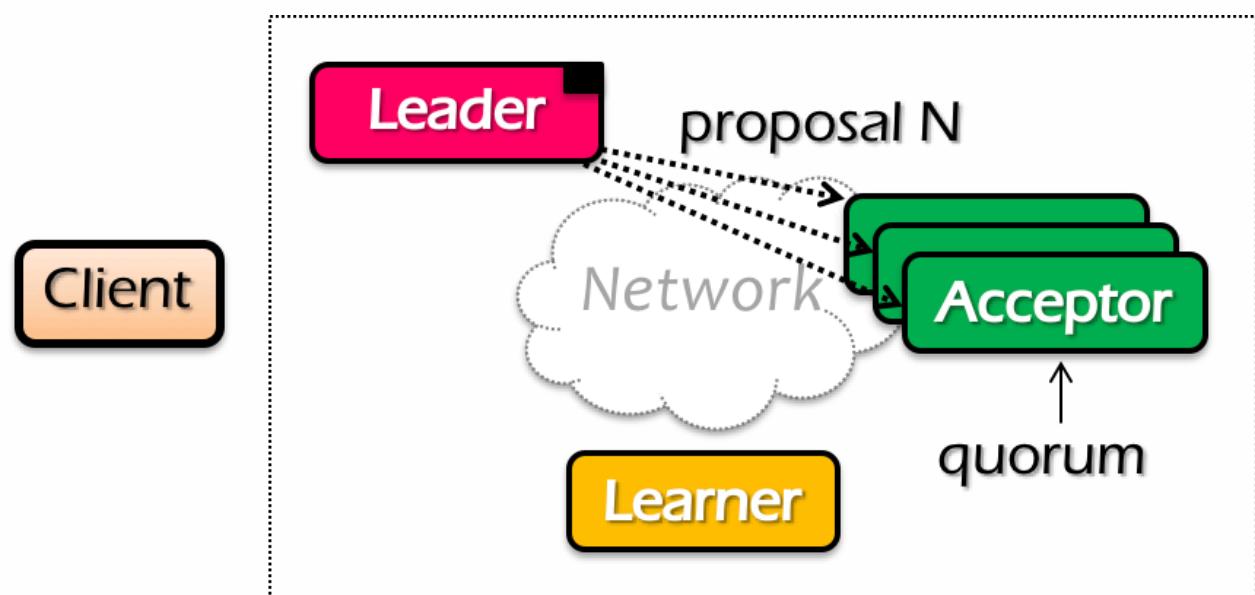


Procedure

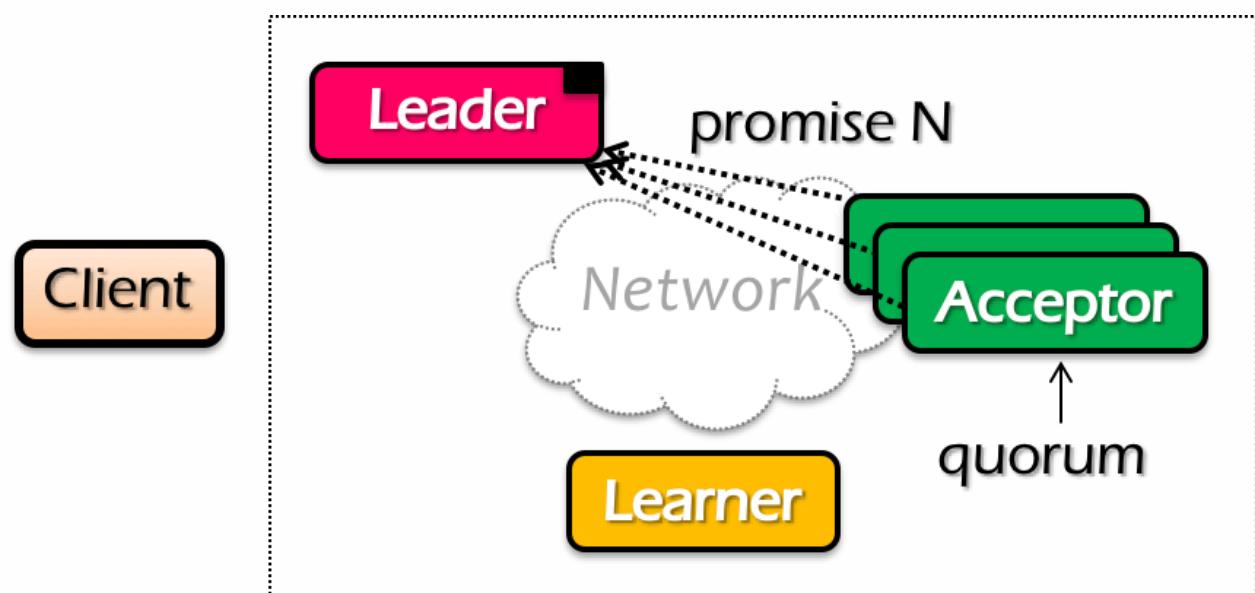
【Phase 0】首先，Client 还是地给 Proposer 发送请求。



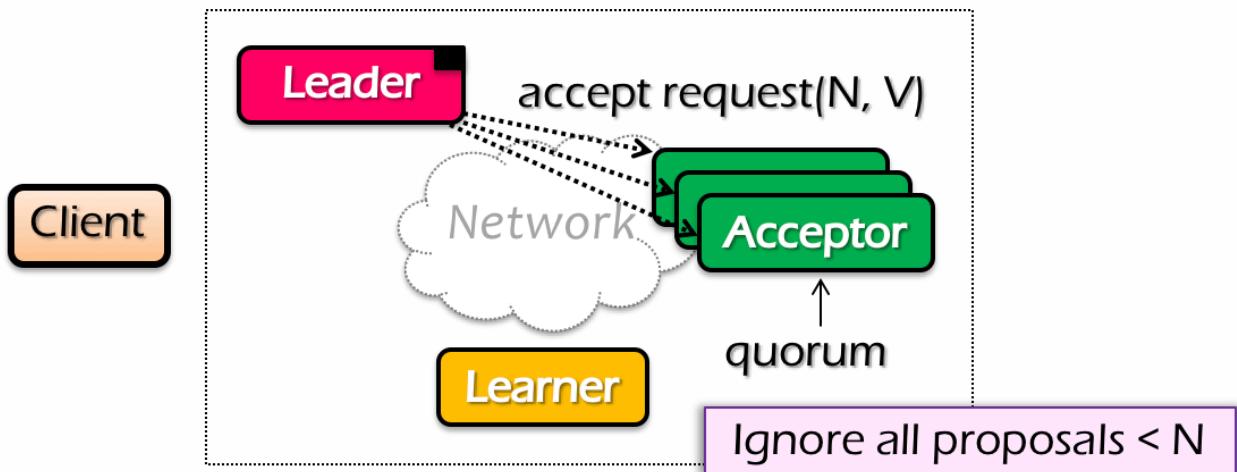
【Phase 1a】Leader 就创建一个最新的 Proposal ID（越大越新）并把它发送给 Quorum。



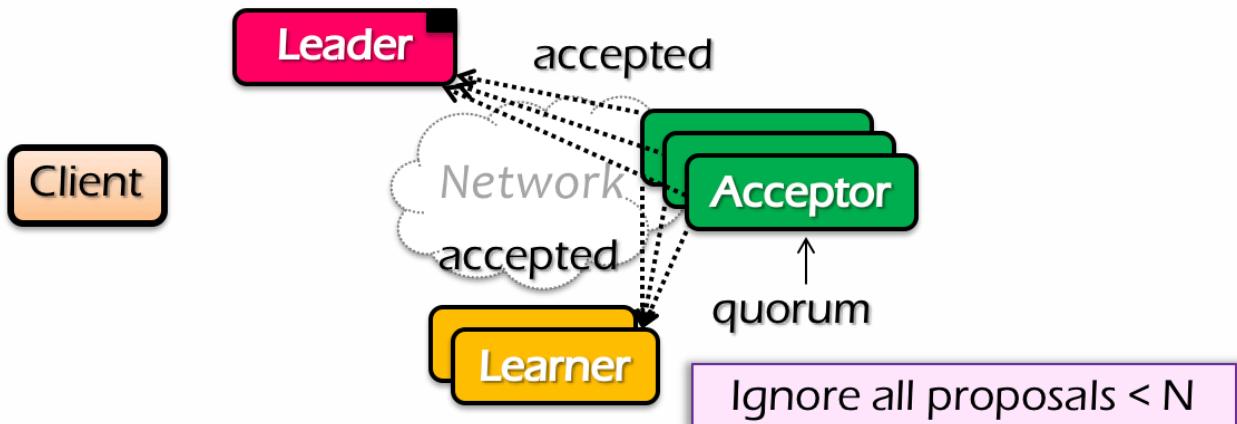
【Phase 1b】Acceptor 来判定 Proposal ID 是否是最新的。如果这个 ID 比曾经见到过得还旧，那就丢弃它。否则，把这个最大 pID 及其对应的值返回，并且更新自己的记忆。



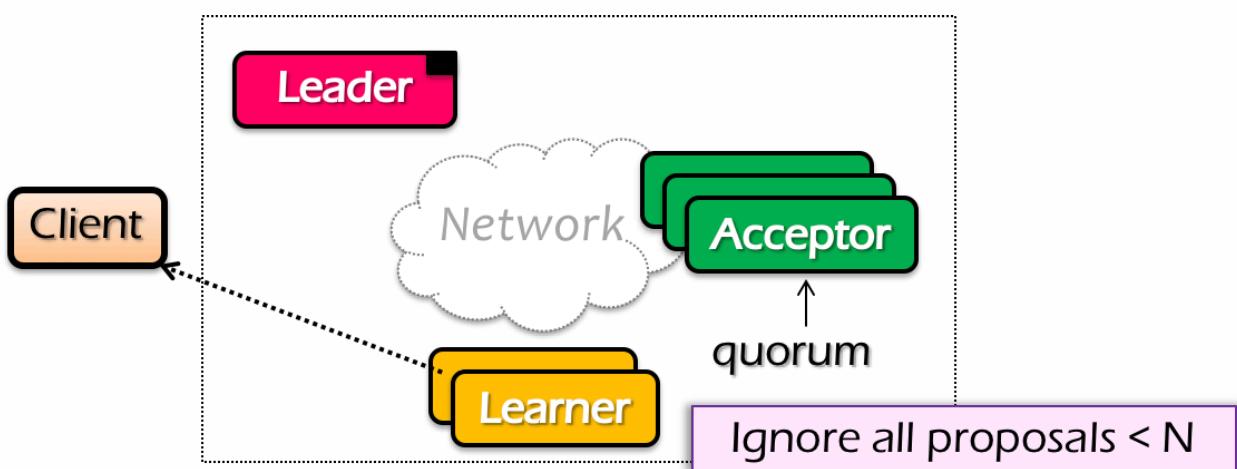
【Phase 2a】 Leader 会从 Acceptor 中收集 Promise。假如收集到的数量足够，就把值 V 放到 Proposal 里面去。然后，发送 Accept Request 连带刚刚拿到的值 V 给 Quorum。



【Phase 2b】 Acceptor 会判断 Promise 是否仍然被持有。假如是的话，就注册值 V，并且发送一个 Accepted Message 给 Proposer 跟 Learners。假如不被持有，就忽略这条消息。



【Phase 3】 Learner 会给 Client 一个回应，与此同时可能根据 Request 采取一些行动。



CAP Theorem

Expression

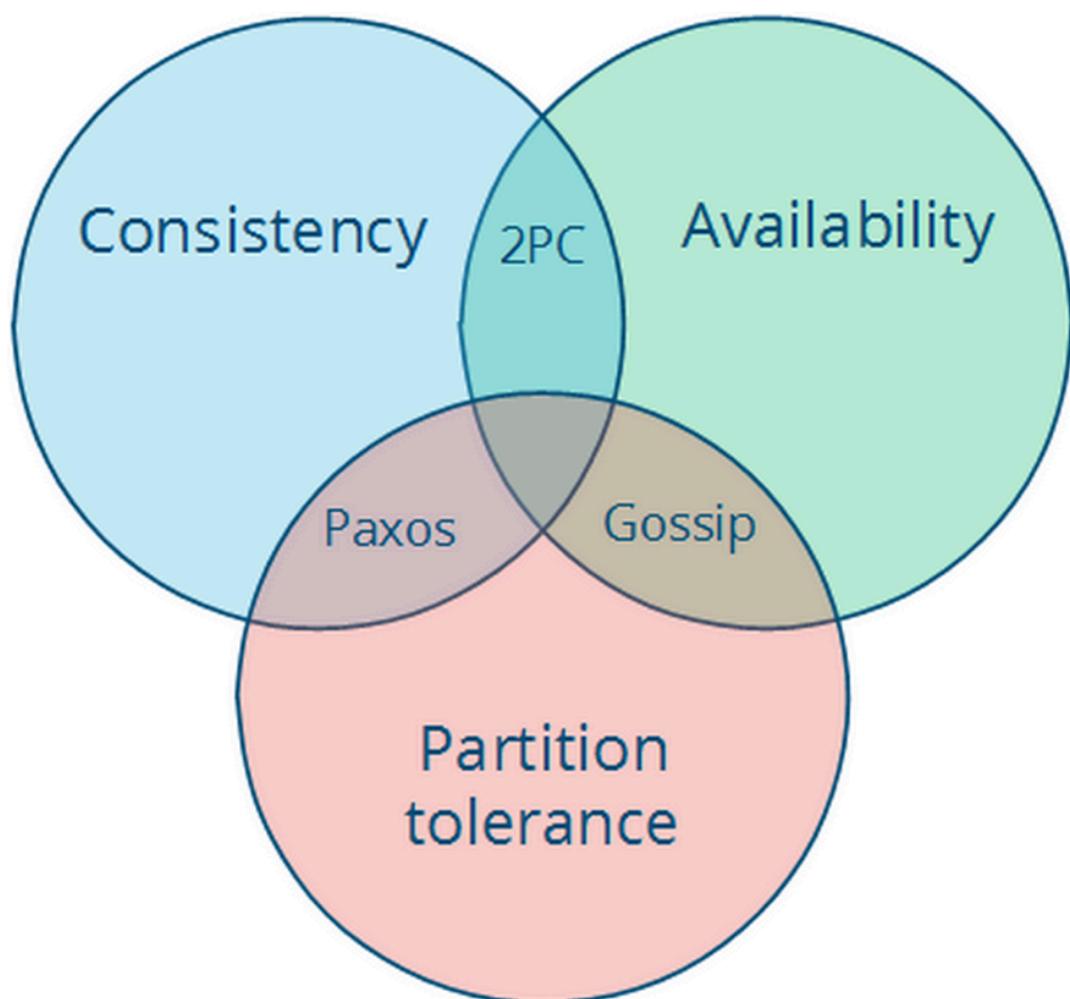
分散式運算機系統不可能同時提供以下所有三項保證。

- 一致性（所有節點同時查看相同的資料）
- 可用性（保證每個請求都會收到有關其成功還是失敗的回應）
- 分區容差（系統繼續運行，儘管系統部分出現任意消息丟失或故障）

这个我们之前也提到过。

Balancing

在它们三者之间做权衡，你会得到这么一个 3R 图：



Lecture 21: P2P & Blockchain

P2P

Point to Point...

意味着所有的功能都是由网络中普通的结点交汇沟通实现的，不存在一个包办一切的家长服务器。

但我们仍然希望能够没有中央伺服器的情况下保证：

- 结点的跟踪和查找
- 分布式存储
- 防止资料丢失
- 保证可用性
- 保证一致性
- 保证安全性
- 保证匿名性

.....这就只能靠高超的智慧了。

幸好有些人很聪明。

Example

BitTorrent 应该是最经典的 P2P 例子了。

一个 Torrent 文件仅仅是一段短短的字符串：

```
{  
  'announce': 'http://bttracker.debian.org:6969/announce',  
  'info':  
  {  
    'name': 'debian-503-amd64-CD-1.iso',  
    'piece length': 262144,  
    'length': 678301696,  
    'pieces': '841ae8.....8190a4'  
  }  
}
```

保存着 Torrent 文件的声明人（Announcer），文件名、大小、块大小、以及每块的 Hash 值（SHA1s）。

并没有包含任何「你可以去访问那个域名下载到这个文件」的知识。

为了下载一个 BitTorrent 文件，你就不得不去一群 Peers 组成的 P2P 网络，然后请求这个文件（的一部分），拿到每一块之后比对其 Hash 值来确认下载的正确性。

不是所有的 Peer 都拥有所有的块的所有分片。很有可能他所持有的只是很小的一片。但你为了下载整个文件，不得不遍寻 Peer 以拼凑出整个文件。

非常之累。

Traverse Nodes

网络上每个节点都是（相对而言）独立的。不像大公司的服务器，他们之间并没有一个完整的链连接他们。

那么，有什么办法可以遍历整个 Peers 网络呢？

Succesors List

每个结点都持有 r 个下家的 IP 位址。在一定概率上，保证你能通过少数几次遍历找出可用的结点。（ r^n 指数增长。）

当然这并不是一定的，很可能遇到一些垃圾结点（不可用，也没有任何下家信息），也可能运气不好找了好多也没找到。

BitCoin & BlockChain

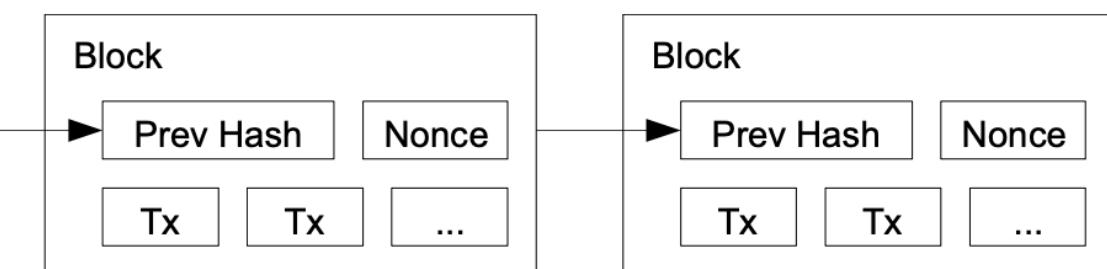
Basis

为了开始使用 BitCoin Wallet，你只需要一对密钥就好了。

一枚公开，用来标识自己；一枚私藏，用于签名。

Idea

信任大多数人 / 信任大多数算力。



Block Chain 的增长是一环扣一环的。如果要修改之前的结点，则必须修改从该节点直到现在的所有后续节点。

主要原因是 Hash 是连续进行，依次依赖的；任何一点微小的改变都导致 Hash 值的巨变、Hash 值的 Hash 值的巨变...这就导致动手脚几乎完全不可能。

假如大部分的 CPU 资源都是诚实的（按规矩办事），那么就保证最长的链总是可以被诚实者掌控。

Trust

意在说明没有任何人值得信任。

如果有可信任的第三方，就不需要区块链了。

代价也是有的。它速度极慢，使用困难。

但那没办法。就是这么设计的。

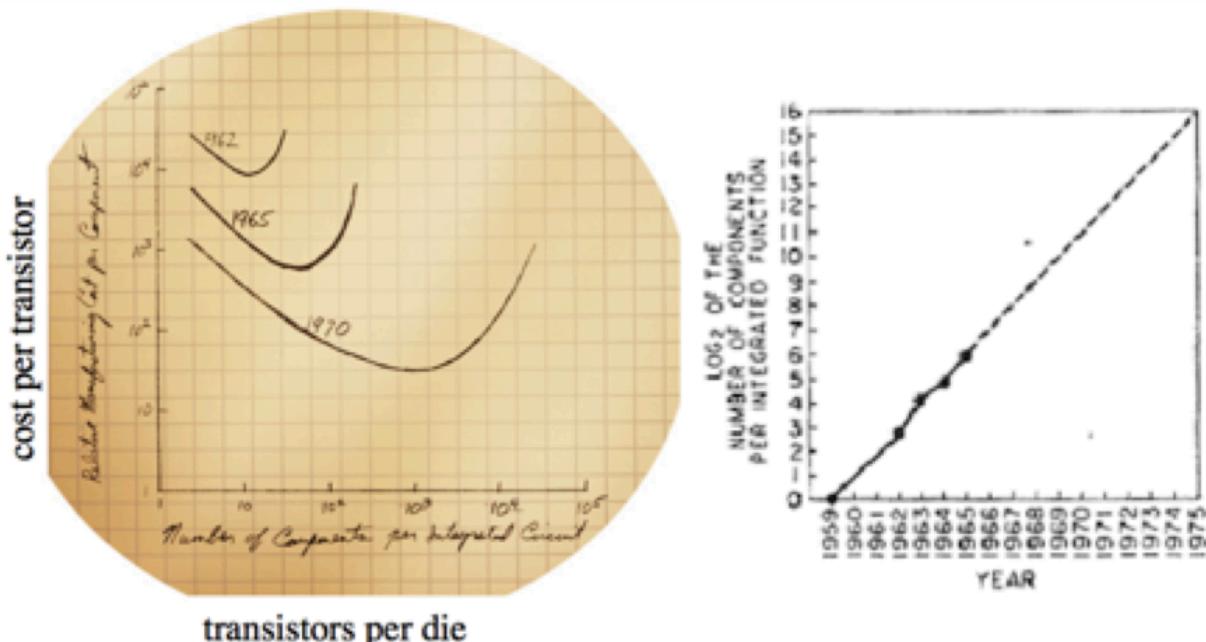
Lecture 22: Performance

| 好想提升效率啊

Bad Performance: Why?

Hardware

花钱买新设备总还是可以的。至少到目前为止，新硬件的性能总还是在提高。



“Cramming More Components Onto Integrated Circuits”, *Electronics*, April 1965

| Moore's Law 说连续翻倍可能夸张了点。

Software

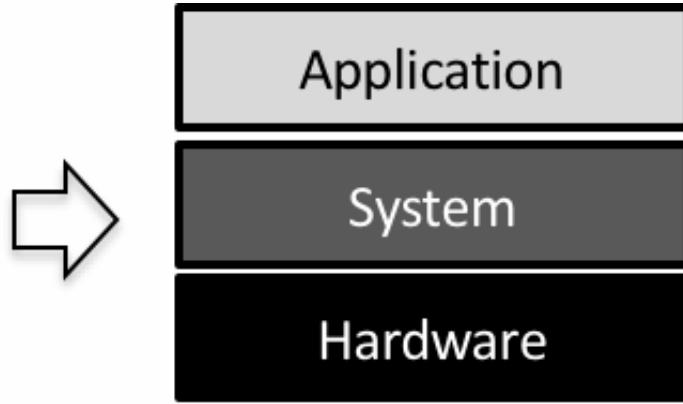
感觉性能成问题，多半是你系统设计垃圾，代码写的差，欠优化。

那还是从软件上解决一下吧。

Find the Bottleneck

优化不是漫无目的的。首先我们要对照系统设计架构，结合时间测量器，测出每一部分对性能的总体影响占比，然后对大头（也就是瓶颈）进行针对性的优化。

这样优化工作的效率是最高的。



相对粗粒度一点的分块，就是这三类。要么砍软件功能，要么优化系统结构，要么买硬件。
你自个选吧。

Performance Metrics: How?

衡量性能

上面的第一步：怎么衡量每一个部件对整体的效率影响？

四部分。

- Capacity
- Latency
- Throughput
- Utilization

Capacity

Capacity 指的是总体的「能力」。

例如，内存设备的容量、处理器的频率、等等...都是 Capacity 的范畴。

Latency

延迟：从拿到请求到完成请求这一过程的耗时。

鉴于信息在系统组件中传递需要耗时间，因此我们有

$$\text{Latency}(A+B) \geq \text{Latency}(A) + \text{Latency}(B)$$

Throughput

吞吐量：所做的事情（量化为 Capacity）和所耗的时间之比。

因为信息传递的耗时，以及串行系统的阻塞性也可以知道

$$\text{Throughput}(A+B) \leq \min(\text{Throughput}(A), \text{Throughput}(B))$$

系统中吞吐量最小的那个节点就是制约整体吞吐量的那位了。

我们尊称之为 Bottleneck。

Utilization

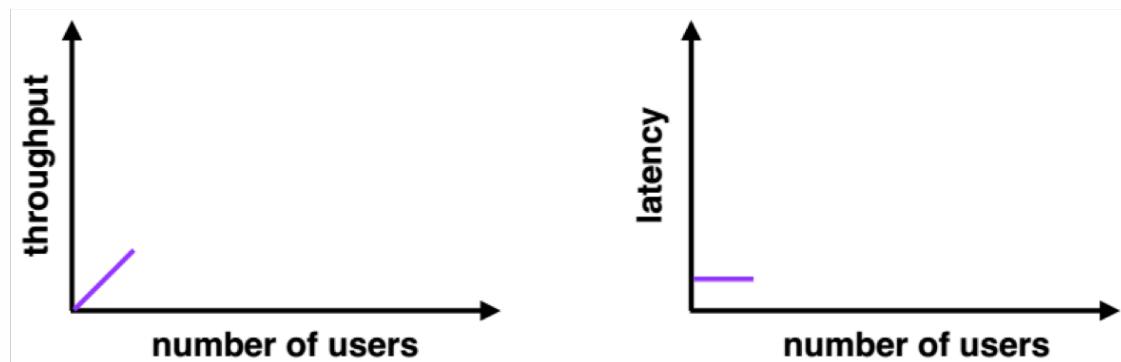
实际被使用到的 Capacity 占设备总体 Capacity 的比值，称之为利用率。

这个很直观，CPU Cycle 数、内存占用率、磁盘占用率等等都是。

Relationship

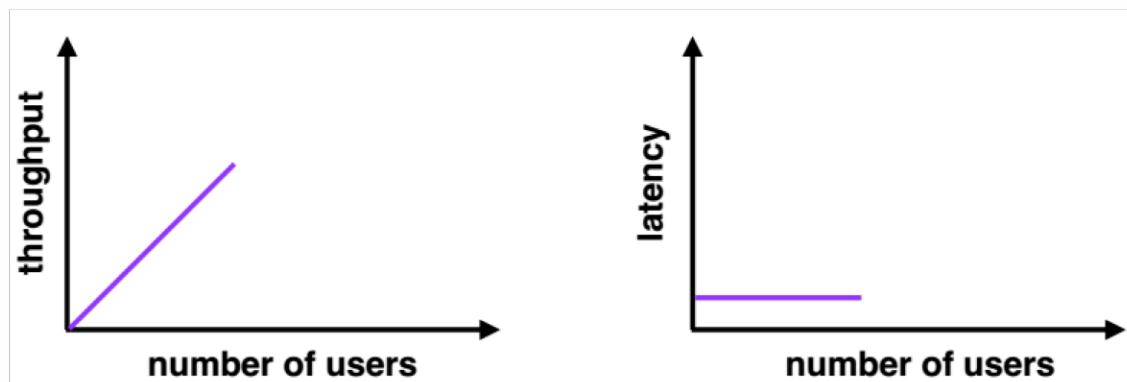
Throughput and Latency

- Few users
 - Low latency
 - Low throughput (few users = few requests)



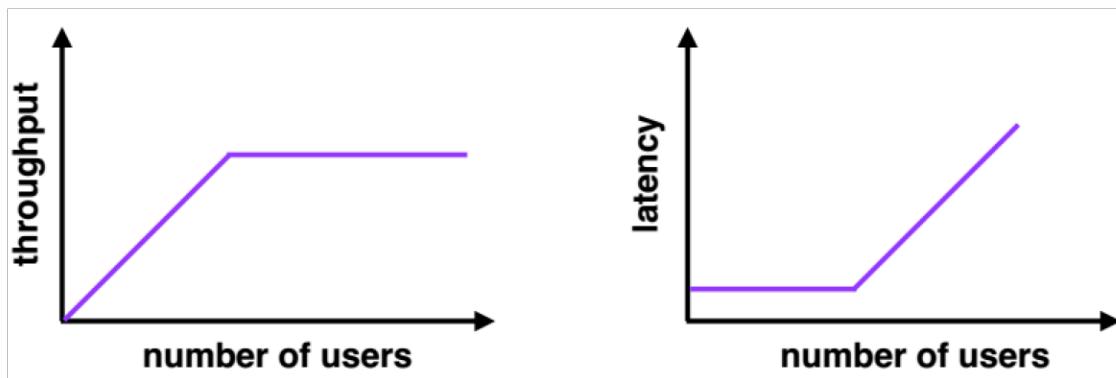
用户很少的时候，Latency 保持在较低的值，Throughput 均匀增长（此时制约量为需求数。），Utilization 很低。

- Moderate users
 - Low latency (new users consume previously idle resources)



这种状态在系统尚未满载的时候持续...

- Many users
 - High latency (requests queue up)
 - Throughput plateaus (cannot serve requests any faster)



在资源饱和之后，Throughput 不再增长了（因为 User 的增加反过来导致 Latency 增加，结果 Throughput 涨不起来）。

Fighting with Bottlenecks: Yay!

Where

通常来说，造成瓶颈的因素很多。

- 物理限制
 - 光速有限 (.....)
 - 内存、磁盘有限
 - CPU 主频有限
- 共享
 - 多个用户共享一台设备
 - 多个客户端共享一台服务器

How

假设我们现在找到了 Bottleneck 的所在。那么我们能怎么修正呢？

Solution 1: Batching

通常来说，把细碎的小操作打包成一个大操作都能节约边际成本。

Solution 2: Dallying

| Dally 差不多就是 Delay 的意思，磨磨蹭蹭。

比如，本来要立即写入磁盘的文件，拖着不写，而是放在 Temp 里；Caches 也是这种概念的体现。

Solution 3: Speculation

猜测。猜你会做的事情并且提前做好，你真的决定要做的时候就会很快。

比如，Processor 的预测指令执行，File System 的 Prefetch 等等，都是 Speculation 的体现。

How...ever

然而，上面三种 Solution 都毫无疑问地引入了复杂性。

不说你算法写的不好的话可能还会降低性能，

这些复杂性会带来大量的 Concurrency 问题。

谨慎使用，好吧。

Lecture 23: Performance II

今天主要讲讲调度对于性能的影响。

Scheduling

计算机系统的每一层抽象里，都跟调度分不开。

如何分配处理器的时间？我们引入了线程调度。

如何分配 RAM 空间？我们引入了内存地址空间。

如何分配打印机资源？我们引入了打印任务。

如何分配磁盘？我们引入了磁盘请求。

如何分配网络资源？我们引入了 Packets。

如何分配内存总线？我们引入了内存请求。

总之，调度是不可以没有的，否则就会带来混乱和延迟。

Difficulties

大家都知道，最难的点是如何在资源不足的情况下调度。

要是资源数多于请求数，那太简单了，每人一个都还有剩。

然而在资源不足的情况下，你还缺少具体的信息、缺少办法来强制回收资源（自己也建立在这层抽象上面），真实困难重重。

Ideal Scheduler

理想的调度器么…

- 延迟小。就是你最好少干点事情，否则给你卡半天大家不要干活了)
- 吞吐量大。这个跟上面一个意思（
- 资源占用低。不希望一个资源调度器自己占太多资源（？）
- 公平性。每个请求都有基本相同的概率获取资源，而非被少数垄断。
- Scalability。最后，希望在压力增大的时候不要显著降低吞吐率，而是将其保持在一个水平不变。就像上面的图一样最好。

留意到，这些要求并不是都能实现，在某些情况下甚至相互矛盾。

但我们应该分情况讨论，尽量设计出最好的调度器。

Measuring Time

我们如何一个请求的处理时间？三种方式：

1. **Response time** – Response time is the time taken to start responding to the request. A scheduler must aim to minimize response time for interactive users.
2. **Turnaround time** – Turnaround time refers to the time between the moment of submission of a job / process and the time of its completion. Thus how long it takes to execute a process is also an important factor.
3. **Waiting time** – It is the time a job waits for resource allocation when several jobs are competing in multiprogramming system. The aim is to minimize the waiting time.

Copied from [here](#). 比起来 PPT 里完全不是人话。

Response Time：从请求发出开始要经过多长时间才能进行响应（被调度到）？

Turnaround Time：从请求发出开始要经过多长时间才能完全完成？

Waiting Time：在产生资源竞争的时候，单个线程要等待多久？

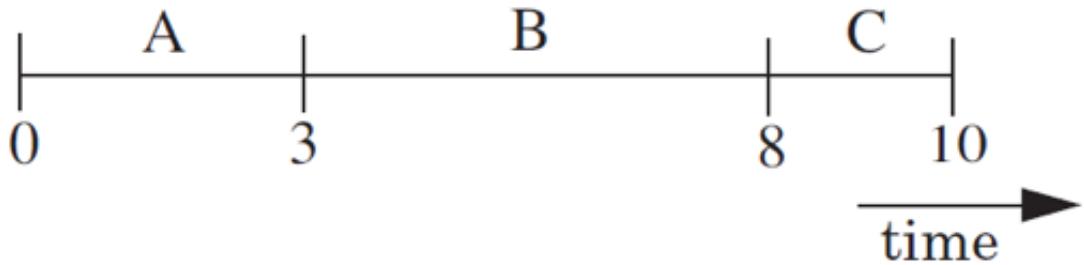
Scheduling Strategies

主要的调度策略包括：

FCFS

也就是 First Come First Server。

Example



Job	Arrival time	Amount of work
A	0	3
B	1	5
C	3	2

到得早的人先享用资源。

Job	Arrival time	Amount of work	Start time	Finish time	Waiting time till job starts	Wait time till job is done
A	0	3	0	3	0	3
B	1	5	3	8	2	7
C	3	2	8	10	5	7
Total waiting						7

Analysis

首先这个算法是公平的。

但是因为太简单（贬义），我们一般只把它用在打印机调度程序上。

Convoy Effect

在使用 FCFS 的资源调度里，我们有一种很特别的现象：Convoy 现象。

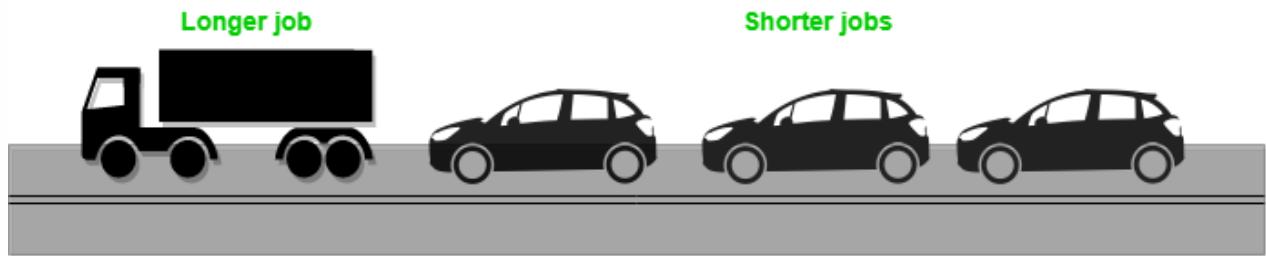


Figure - The Convey Effect, Visualized

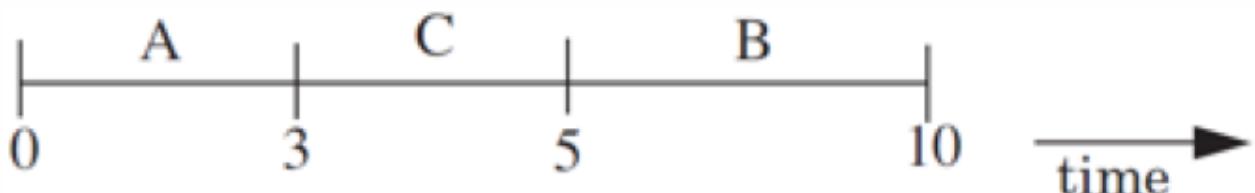
- The I/O bound processes are first allocated CPU time. As they are less CPU intensive, they quickly get executed and goto I/O queues.
- Now, the CPU intensive process is allocated CPU time. As its burst time is high, it takes time to complete.
- While the CPU intensive process is being executed, the I/O bound processes complete their I/O operations and are moved back to ready queue.
- However, the I/O bound processes are made to wait as the CPU intensive process still hasn't finished. **This leads to I/O devices being idle.**
- When the CPU intensive process gets over, it is sent to the I/O queue so that it can access an I/O device.
- Meanwhile, the I/O bound processes get their required CPU time and move back to I/O queue.
- However, they are made to wait because the CPU intensive process is still accessing an I/O device. As a result, **the CPU is sitting idle now.**

简单说，用这种策略最后不是 I/O 设备闲着，就是 CPU 闲着。总归没法充分利用资源。

这不好。

SJF

Shortest Job First 指的是让（耗时）最短的任务先做！



也就是，在上例中的 ABC 请求会按照这种顺序被调度。

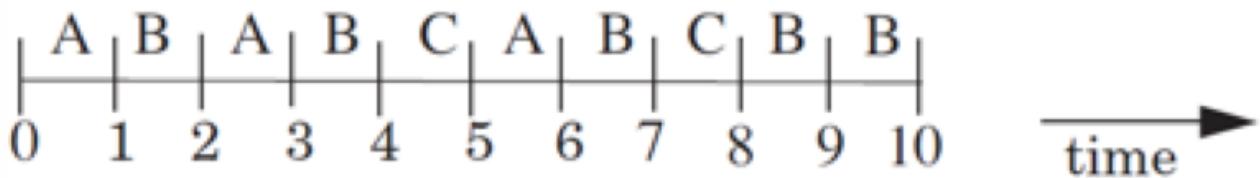
注意这种简单的情况不考虑打断。也就是说，在执行的过程中遇到了一个刚到的耗时更短的任务，不会打断当前任务的。

调度器只会在当前活跃任务结束的时候按照时间排序，决定谁来进入下一步调度。

Job	Arrival time	Amount of work	Start time	Finish time	Waiting time till job starts	Waiting time till job is done
A	0	3	0	3	0	3
B	1	5	5	10	4	9
C	3	2	3	5	0	2
Total waiting					4	

RR

亦即 Round Robin 算法。



每隔一段时间就打断目前正在执行的任务（这里是一秒），并轮流地引入下一个任务。

Job	Arrival time	Amount of work	Start time	Finish time	Waiting time till job starts	Waiting time till job is done
A	0	3	0	6	0	6
B	1	5	1	10	1	9
C	3	2	5	8	2	5
Total waiting					3	

PSP

Priority Scheduling Policy。按优先级调度策略。

给每一个到来的任务都分配一个优先级（可以是动态的，也可以是静态的）。

每次调度，都选出优先级最高的任务运行。

现代 CPU 的调度策略是基于优先级和 Round Robin 的混合调度。

性能和安全...如何权衡？

Meltdown

Intel 著名的「炉芯熔毁」漏洞就是因为一类针对 Cache 的优化使得面对不同输入的执行时间不同，结果可被用于探测内存分布的问题。

Lecture 24: Security Intro

安全性导论...

接着上节课的 Performance 跟 Security 的权衡说。

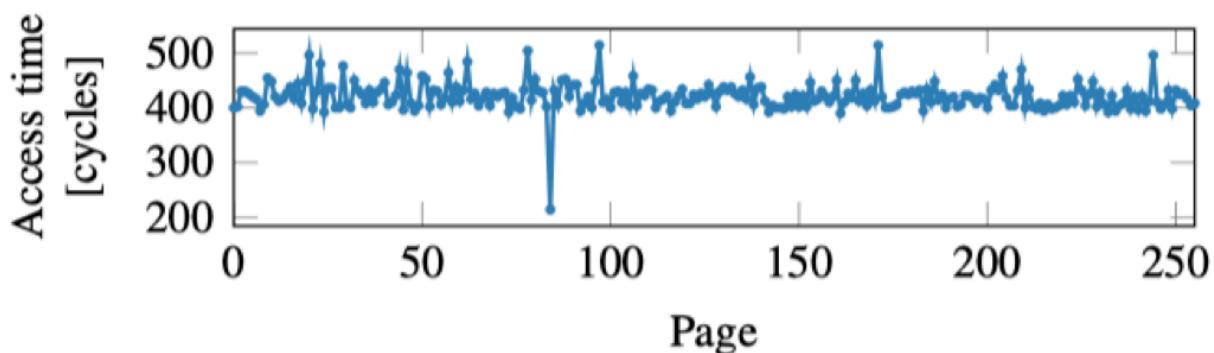
Reload Attack

所谓重放攻击，是非常流行的一类攻击手段。

Flush + Reload Attack

这里讲一下 Flush + Reload 攻击：基于缓冲区 Flush 和重放的一类攻击手段。

这种攻击手段可能会用于泄露信息，通过针对不同内存页的响应时间来确定其被保存在哪一页中。



看起来这种手段还是很有效的。

Lessons

我们可以学到，共享任何东西都是不安全的，很有可能造成信息泄露和风险，有时候还是以一些奇怪的方式。

但如果我们拒绝共享，追求 Isolated，那对性能的影响更加不能接受。

所以...也是一个程度的问题。

Attacks

世界从来都不安全。互联网的世界更是如此。

选择将电脑接入互联网，就是把自己的计算机暴露在整个的危险空气里。不知道有什么消息会过来，不知道自己的计算机有什么漏洞。云云。

Dangerous

事实上，保证安全是很难的，而攻破防线却很简单。

创造出有价值的东西是很难的，而要毁灭却很简单。

而且，「安全」永远是一个消极的词汇。他的意思是「没有人可以……（比如，获取你的信息，改写你的文档，等等。）」。

这是一个全称否定命题。要保证他的正确性是很困难的。

而且，只要有一点例外不满足，整个命题就全毁。

也就是说，在安全上，Fault 是不能被容忍的。

Real World

要确认「…不能…」比起确认「…能…」要困难得多。

这个世界就是这么回事。

Security Goals

我们在讨论安全的时候，到底在讨论什么？

为了精确描述我们的意图，我们换一些用词。

Confidentiality

秘密性：确保谁（不）能读到数据。

Integrity

完整性：确保谁（不）能写入数据。

Threat Model

在真实的攻防中，我们作为「守」的一方，对于攻方究竟会做什么是一无所知的。这就使得防守十分困难……

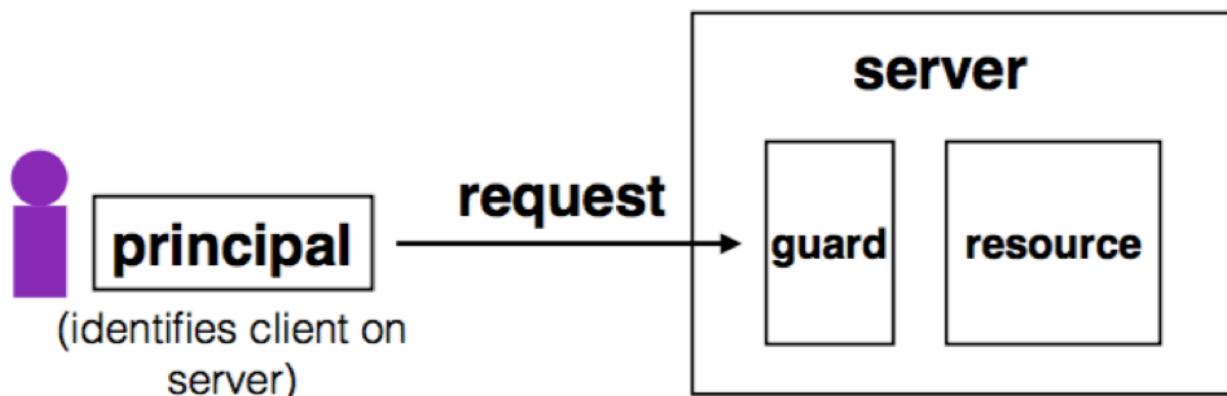
攻方并不需要怕犯错（大部分时候）。他总是可以重试。而一旦他成功了，防守也就完全失败。

要分析威胁模型是很困难的……

Guard Model

防守模型相对明了、正经一些。我们来分析一下这个。

Complete Mediation



所有的 Client (包括恶意的) 的请求都会先进入 GUARD (看门的) ，由它来分析 Authentication 和 Authorization 。

Authentication

这个用户...是他宣称的那个人吗 ?

Authorization

这个用户...有权做这件事 (访问 / 写入某资源) 吗 ?

这两个词英文词意很接近，但是表意有区别。一定注意。

Lecture 25: Authentication

Privilege

Definition

我们用「Privilege」来描述一个（一类？）用户所能执行的操作。如果某用户没有这种「授权」，它就无法进行特定的操作。

Don't Trust Anyone

信任是一件很危险的事情。

因为理论上信任链是可以传递的，因此如果你信任的一个组件爆炸了（被 Hacker 控制或是出现了可被利用的漏洞），那就全完了，所有人都能获取这一权限。

因此，授权越少越好，一定杜绝不必要的授权。

Cost of Security

保证安全的开销有多大？

有多大必要这么做？因为有的时候我们真的会做太过，有的时候又做的完全不够。

Lecture 26: Secure Channel, Local Security

Secure Channel

何谓安全通道？

在密码学中，安全通道是一种传输数据的方法，可以防止窃听和篡改。

最容易想的方法就是加密，用秘密语言通信。

Encryption

encrypt(key, message) → ciphertext
decrypt(key, ciphertext) → message

```
encrypt(34fbcb1, "hello, world") = 0x47348f63a67926cd393d4b93c58f78c  
decrypt(34fbcb1, "0x47348f63a67926cd393d4b93c58f78c") = hello, world
```

property: given the **ciphertext**, it is (virtually) impossible to obtain the **message** without knowing the **key**

MAC(key, message) → token

```
MAC(34fbcb1, "hello, world") = 0x59cccc95723737f777e62bc756c8da5c
```

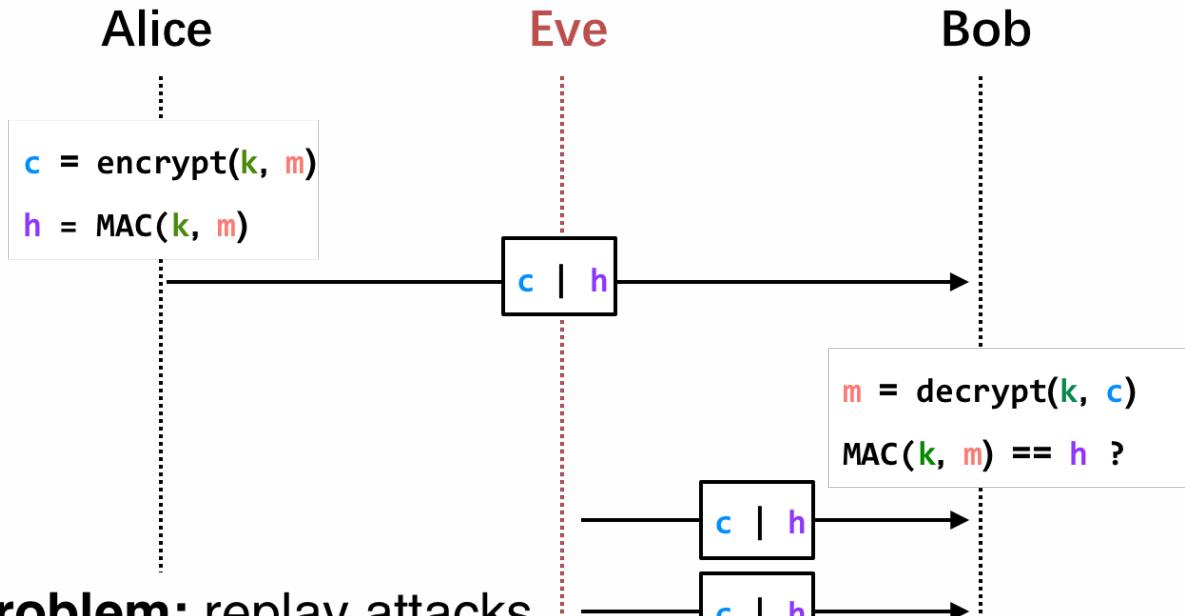
property: given the **message**, it is (virtually) impossible to obtain the **token** without knowing the **key**
(it is also impossible to go in the reverse direction)

留意到 encrypt 和 decrypt 是互逆运算，只需要持有一个 key。

然而 MAC 生成出来就不是给你解密的。它保证的是除非你知道 key，否则不可能生成出对应的 token。

Problems

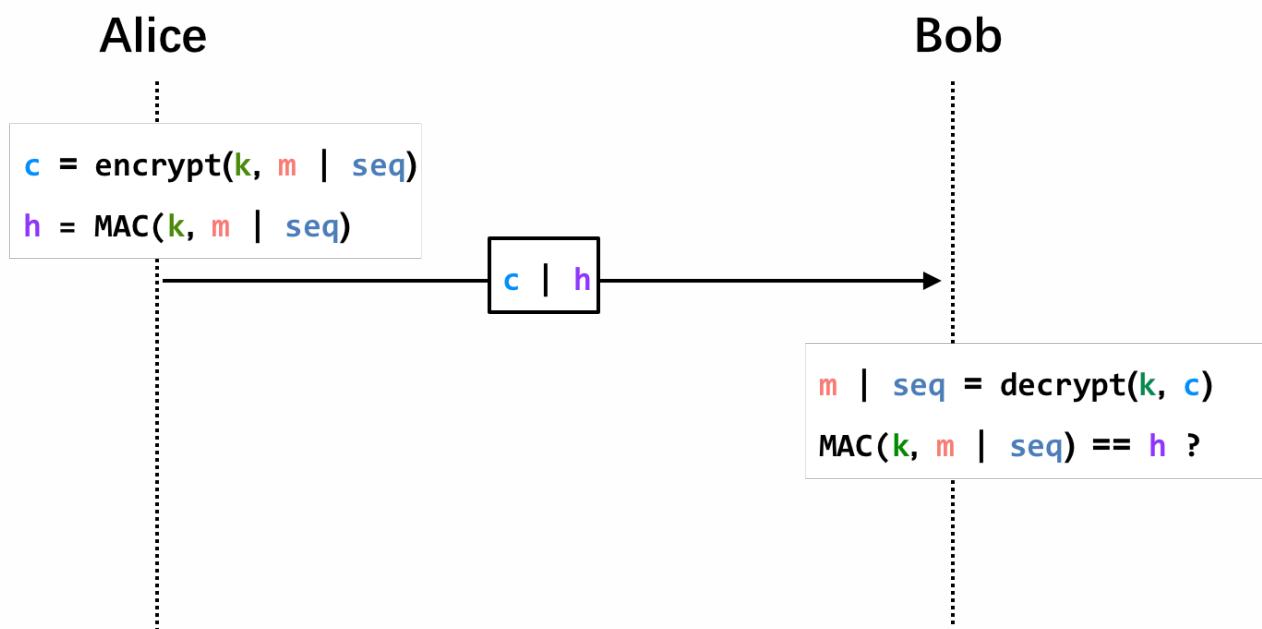
如下图所示，假如我们仅仅用 encrypt 和 MAC 来确保安全信道，就无法阻止重放攻击。



(adversary could intercept a message, re-send it at a later time)

Add a seq

更好的办法是每次生成一个任意的随机信息 seq，跟 Message 一起发过去。这样，每次的 $\text{MAC}(k, m \mid \text{seq})$ 都不一样了。重放就无效了。



Reflection Attack

但是问题是攻击者可以做「反射攻击」：

Alice

Eve

Bob

```
c = encrypt(k, m | seq)  
h = MAC(k, m | seq)
```

c | h

c | h

```
m | seq = decrypt(k, c)  
MAC(k, m | seq) == h ?
```

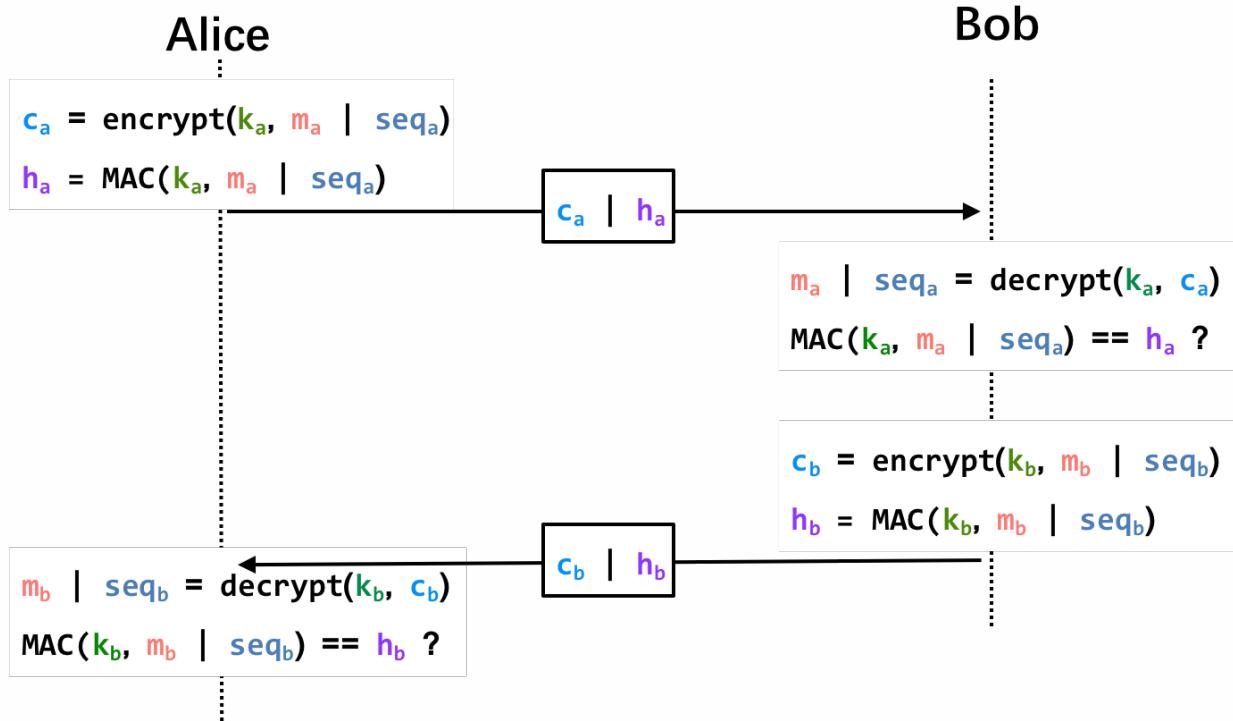
problem: reflection attacks

(adversary could intercept a message, re-send it later in the opposite direction)

把发过去的 $c|h$ 发还给自己，就有可能达成攻击目的。

解决方案也很简单：发送和接收用不一样的 Key 就成了。

Final Solution



Before That

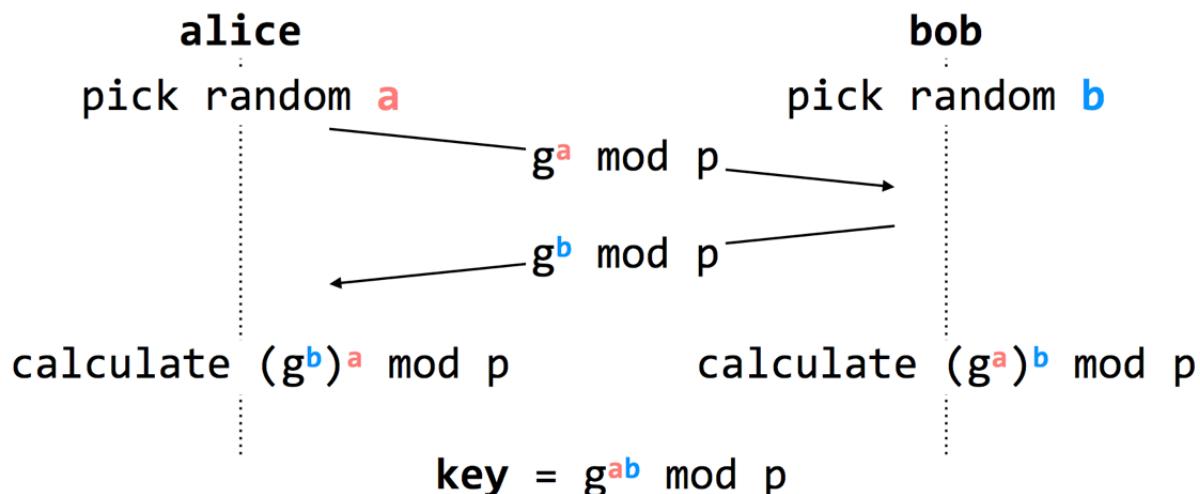
在开始的开始，双方该如何交换他们的密钥，这是加解密所必须的？

Key Exchange

problem: how do the parties know the keys?

known: p (prime), g

property: given $g^r \bmod p$, it is (virtually) impossible to determine r even if you know g and p



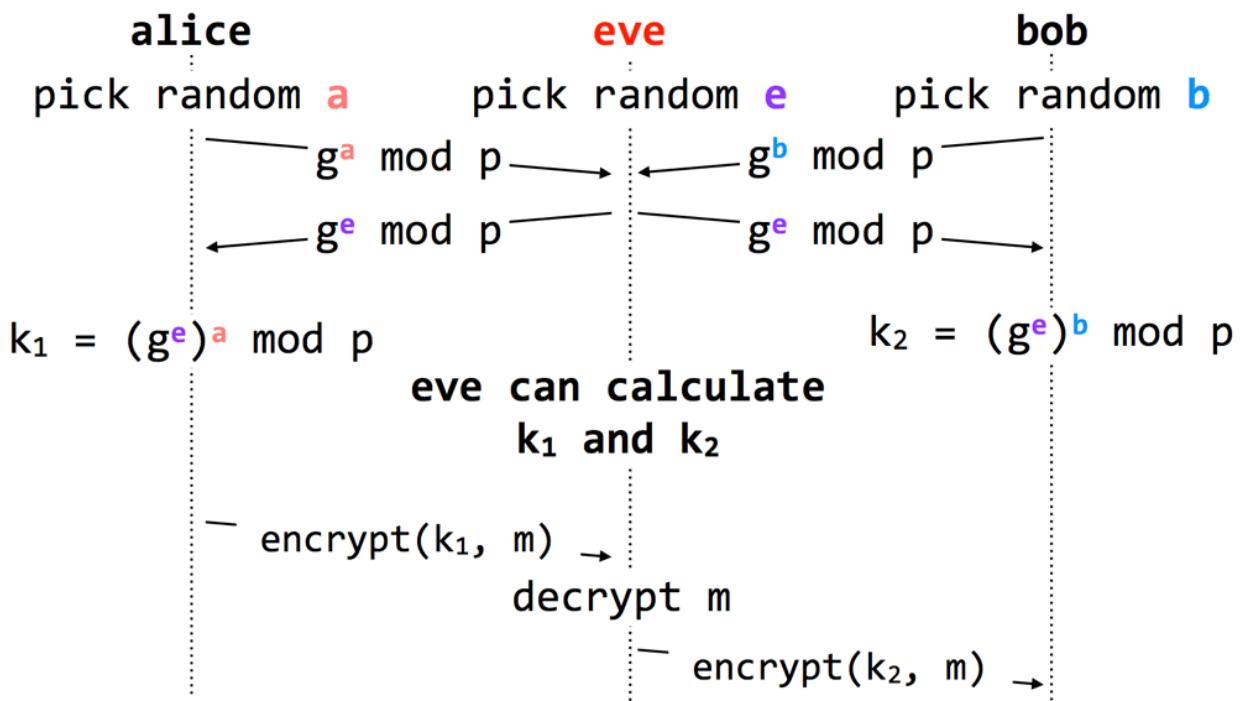
为了在一开始交换双方的 Keys，我们利用了这么一个数学特性：

- 给出一个数字 g 和质数 p ，令 $g^r \bmod p = A$ ，在我们知道 g 、 p 、 A 的情况下都很难算出 r 。

因此 Alice 跟 Bob 可以互相挑选随机数 a 和 b ，分别计算 $g^a \bmod p$ 和 $g^b \bmod p$ 并告诉对方。这样，对方都能知道 $g^{ab} \bmod p$ 的值了。我们就以这个作为密钥；而监听到这个信息的攻击者无法从中推算出 a 或是 b ，也就保证了密钥的安全。

Diffie-Hellman Key Exchange

有一种用心险恶的手段：



problem: alice and bob don't know they're not communicating directly

作为一个中间人直接参与到密钥交换的步骤，并且始终扮演这一角色以便解密他们的通信。

太可怕了。

RSA Algorithm

上面的问题是：即密钥分发的安全性无法保证。

Dual-Keys

RSA 要求两把密钥：分别称为 Public Key 和 Private Key。

他们并不是随便选的，而是满足这个条件：

```
sign(secret_key, message) => sig
verify(public_key, message, sig) => true/false
```

Process

公钥是可以公开的，以任何方式（包括 Insecure 方式）都行，因为目的就是为了让大家都知道。

而 Skey 跟 Pkey 的特性也让一件事情成为可能：验证一个 Message 是否真的来自某个人。

Symmetric & Asymmetric

所谓对称的加密，就是加密和解密的密钥是一致的。

非对称加密，即加密和解密的密钥不一致，但都是成对出现的。

通常来说是以 $M^K = \text{Public} + \text{Private}$ 的做法来拆分。

Local Security

应该不会考（逃

Lecture 27: ROP & CFI

Return-Oriented Programming

Stack Buffer Overflow Attack

回忆一下我们之前做的 ICS Lab 3。

本质上是类似于这种

```
void function(char *str) {
    char buf[16];
    strcpy(buf, str);
}
```

没有边界检查和长度限制的 `strcpy`。这是毒药。

我们可以通过构造过长的字符串来改写栈空间中 `char buf[16]` 之后的内存，最重要的就是放 Return Address 的内存了。

如果能改写这一部分，那么可以跳转到任何内存位置继续执行了。

甚至可以自己在栈上写一些代码，然后跳转到栈上执行（这是最简单的一种方式）。

Data Execution Prevention

也就是防止数据被执行。

他就是告诉 CPU：你别读到什么内存位置都拿去执行啊。

比如你读到一块栈上的内存，就不可能用来作为 Instr 执行啊。那是数据段啊。

- Execute *Code*, not *Data*
- Data areas marked **non-executable**
 - Stack marked non-executable
 - Heap marked non-executable
- Hardware enforced (NX: Non-eXecutable)
- You can load your *shellcode* in the stack or the heap… but you can't jump to it

这样，你精心写在 Stack 上的代码在 PC 被跳进去之后，直接遇到 NX（不可执行）标识位，然后就报错了，你的攻击代码也就执行不下去了。

Code Reuse Attack

也是我们在 ICS Lab 里用过的手段：既然你不让执行我写在栈里的 Code，那我从你的大段程序代码里找出一些我要用的小指令，然后拿来做坏事。

这种手段不用多说，我一辈子也忘不了。

Return-Oriented Programming

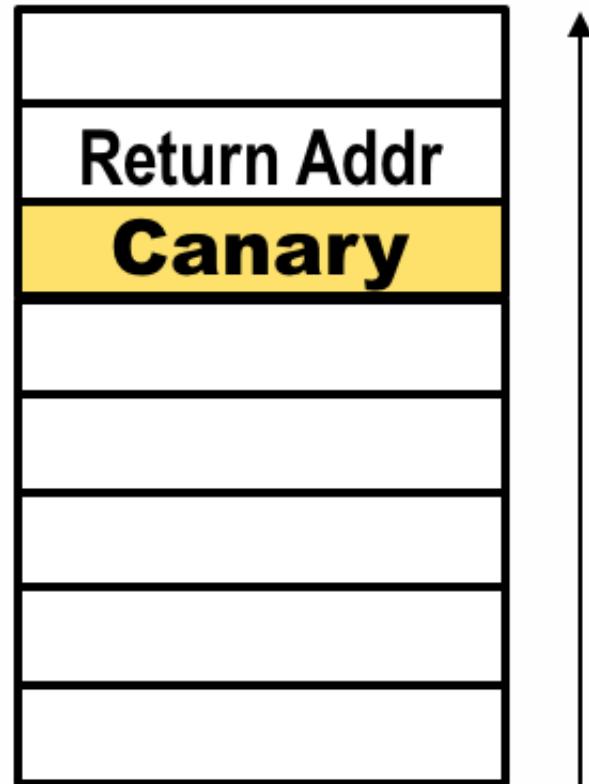
is A lot like a ransom
note, BUT instead of cutting
cut letters from magazines.
YOU ARE cutting out
instructions from text
segments

Defense

防御措施还是很多的，比如：

- 拒绝你看我的 Binary File
 - 比如，作为一个服务器程序运行，骇客就无法返汇编我的二进制；
- ASLR，把代码位置随机化，这样你就无法预料我的代码地址
- Canary，金丝雀法
 - 在栈底部放置一些无用的特定数值，作为金丝雀。假如金丝雀死了（被您的 Overflow 给弄没了），那就说明你改写了栈，拒绝继续执行。

Stack



Control Flow Integrity

| 所谓 CFI，就是怎么保证程序的控制流是我预先规划好的？

CFI: Control-Flow Integrity

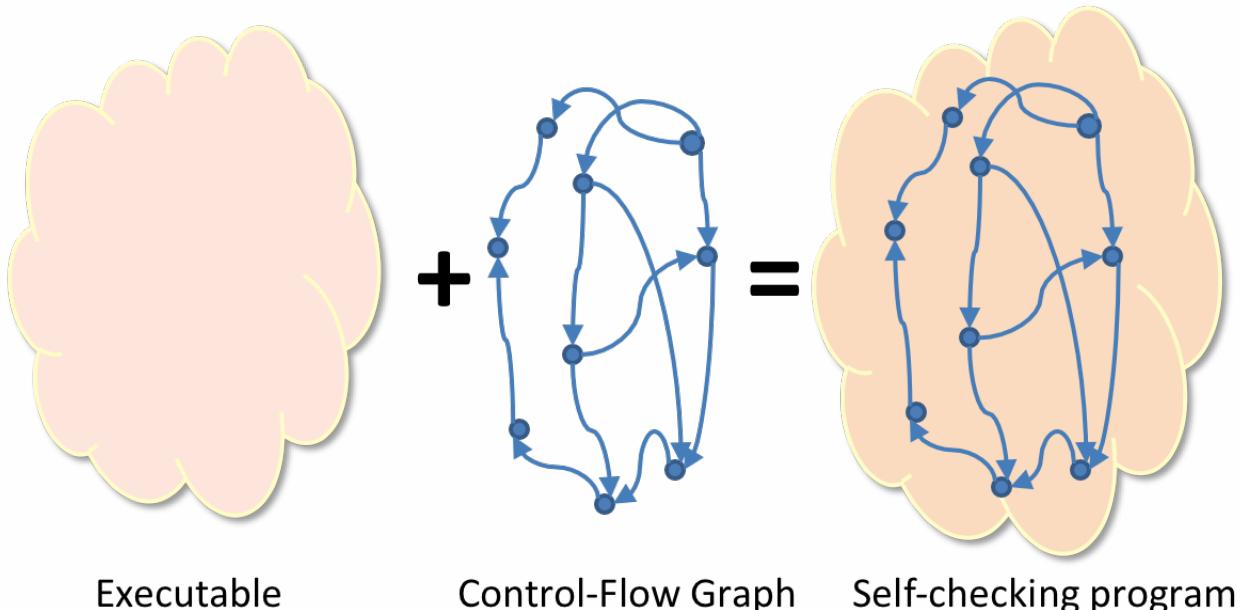
- Main idea: pre-determine **control flow graph** (CFG) of an application
 - Static analysis of source code
 - Static binary analysis ← CFI
 - Execution profiling
 - Explicit specification of security policy
- Execution must follow the pre-determined control flow graph

CFG

| Control Flow Graph

我们先把我们程序的控制流图给他画出来。

这样，我们就可以添加一些检查代码，确认「当我来到这里的时候，我还在这张 Graph 的合理位置上」。



Inside Compilers

Learning Compilers

```
c = next( );
if(c != '\\')
    return(c);
c = next( );
if(c == '\\')
    return('\\');
if(c == 'n')
    return('\\n');
```

```
c = next( );
if(c != '\\')
    return(c);
c = next( );
if(c == '\\')
    return('\\');
if(c == 'n')
    return('\\n');
if(c == 'v')
    return(11);
```

```
c = next( );
if(c != '\\')
    return(c);
c = next( );
if(c == '\\')
    return('\\');
if(c == 'n')
    return('\\n');
if(c == 'v')
    return('\\v');
```

- 1) We wish to add the vertical tab (\v) symbol
- 2) We return its ascii value (11) if the symbol is \v
- 3) We recompile our compiler, and we can now change our implementation to simply return \v

考虑我们加入转义字符 \v 的过程：

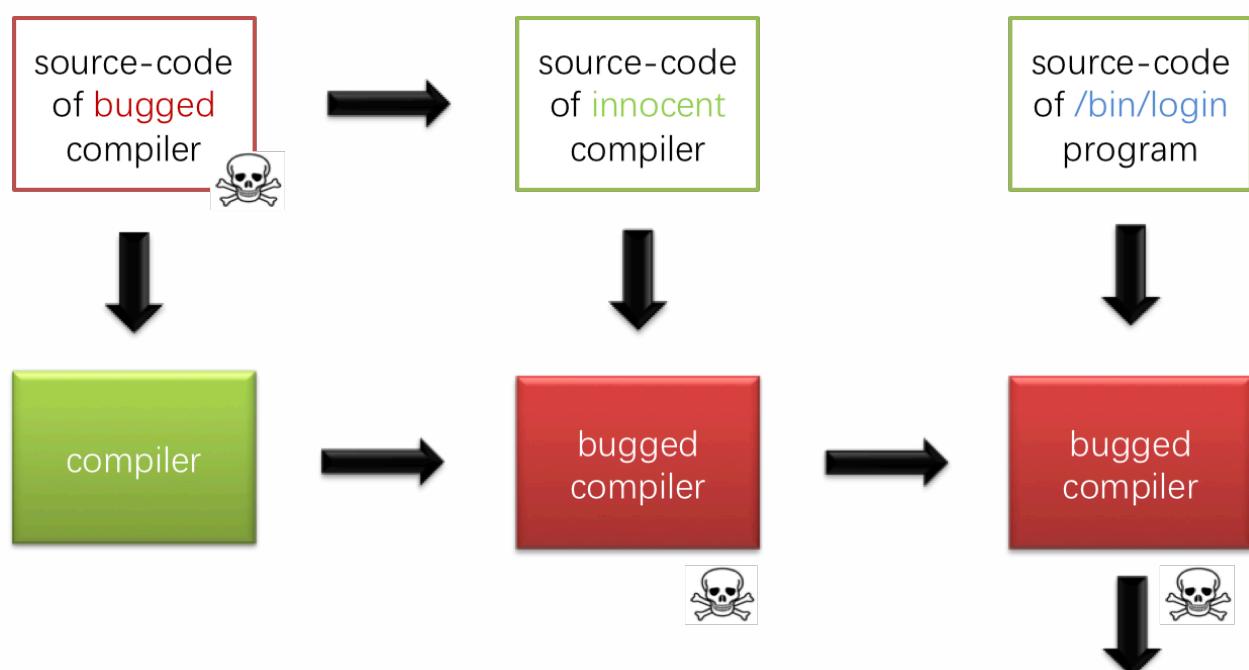
一开始在编译器里，还没有 \v 这个字符，所以我们要手动返回 ASCII 码；然后用旧编译器编译这个新编译器。

现在，这个新编译器认识 \n 了，我们就可以把它改写成 III 的样子了。因为新编译器已经认识 \v 了。

Compile a Compiler

这就提醒我们了，编译器也是编译器编译的。

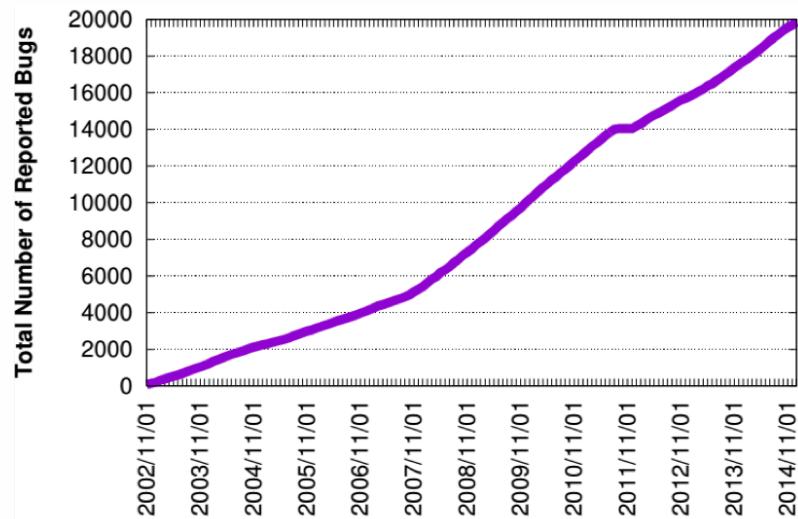
看源代码没问题，不能说明你的程序编译之后还是没问题。



一张图可以说明。

Reconstructing OS

目前，OS 的体量绝对配得上 Monolithic 这个词。



Source: Bugzilla.kernel.com, count of all bugs currently marked NEW, ASSIGNED, REOPENED, RESOLVED, VERIFIED, or CLOSED, by creation date

Linux Bug 的数量也是爆炸式的。

Types of OS

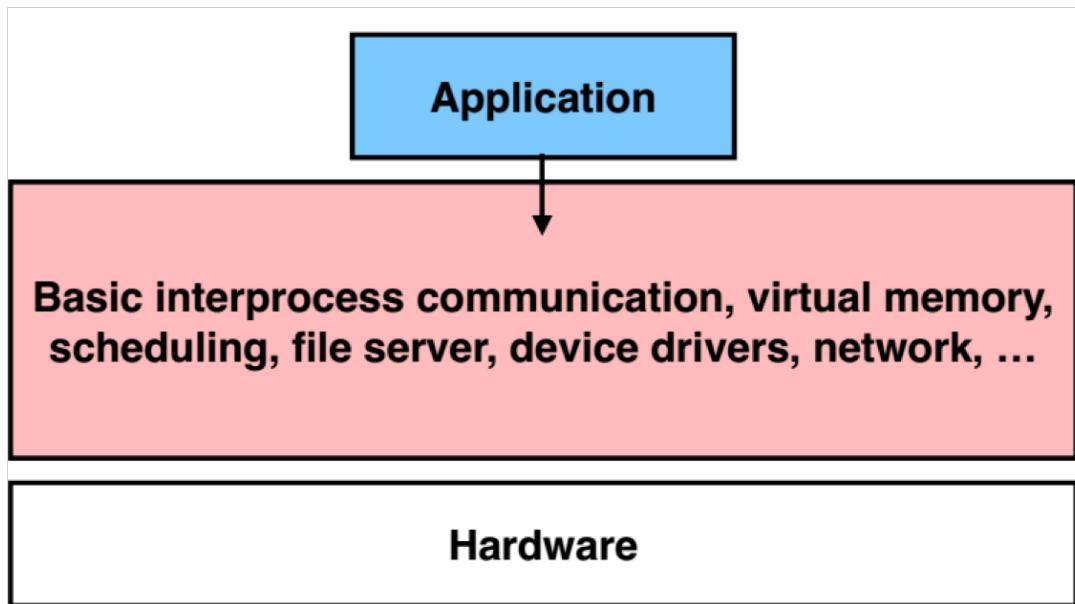
OS 按照内核的构造来分类，可以分为微内核、宏内核和混合内核。

Monolithic Kernel

Monolithic Kernel

- The kernel is trusted for hardware management
 - Memory manager
 - Devices manager, e.g., clock, display, disk
- Modules that manage the hardware are part of the kernel
 - The window manager, network manager , file manager
- Monolithic kernel
 - Most of the operating system runs in kernel mode

Monolithic Kernel: No Enforced Modularity within the Kernel



Pros

- Relatively few crossings
- Shared kernel address space
- Performance

Cons

- Flexibility
- Stability
- Experimentation

Micro Kernel

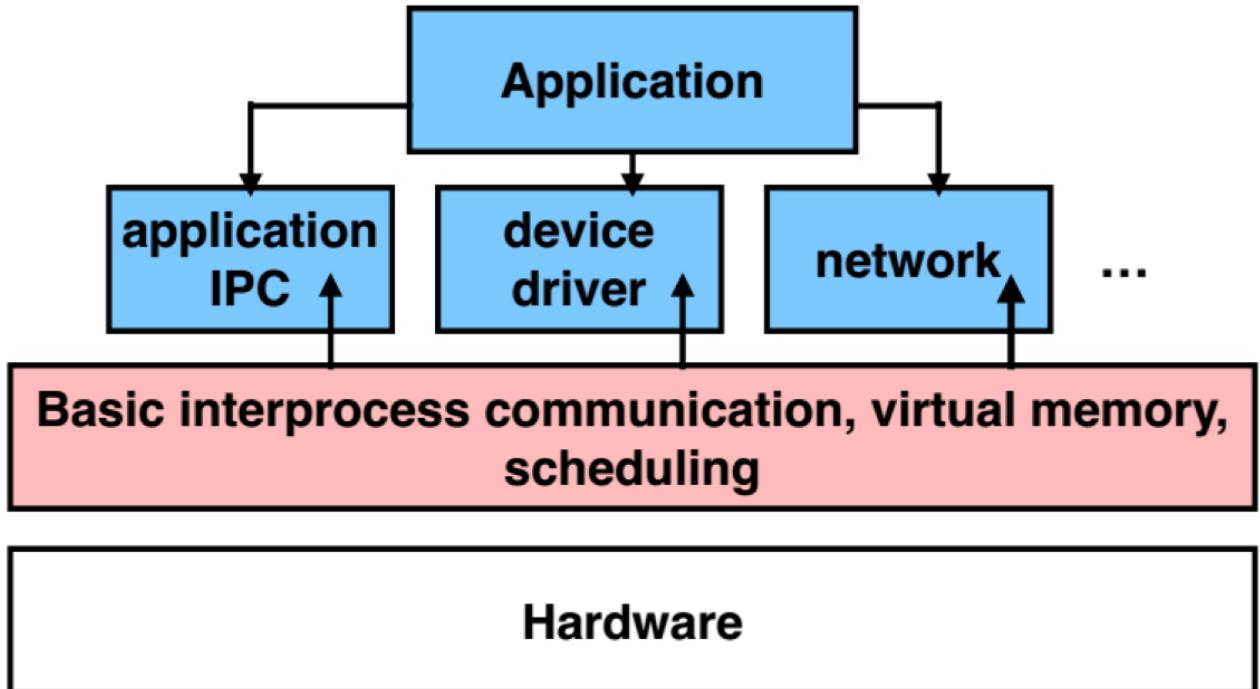
Microkernel

- It is preferred to keep the kernel small
 - Reduce the number of bugs
 - Restrict errors in the module
 - E.g., the file manager module error may overwrite kernel data structures unrelated to the file system
 - Causing unrelated parts of the kernel to fail

Microkernel

- It is preferred to keep the kernel small
 - Reduce the number of bugs
 - Restrict errors in the module
 - E.g., the file manager module error may overwrite kernel data structures unrelated to the file system
 - Causing unrelated parts of the kernel to fail

能不加入 Kernel 的就不要加入。



Pros

- Easier to develop services
- Fault isolation
- Customization
- Small kernel, easier to optimize

Cons

- Lots of boundaries crossings
- (Relatively) poor performance

Root of Trust

信任的根在哪里？

笛卡尔说过，「你唯一不能怀疑的，就是你在怀疑这件事本身。」

Jerome Saltzer 说过，「每个程序都应该只请求最低限度的权限来完成它的工作。」

Xia Yubin 说过，TCB 应该尽可能地小。

TCB: Trust Computing Base

可信计算的根本...根本在哪里？

- TCB is the parts that are trusted
 - Process never trust another process, but trusts all its threads
 - OS never trust a process, but trusts hardware
 - VMM never trust a VM, but trust hardware
 - Which parts do you trust in your laptop? Your phone?
- TCB is the only metric of security
 - Bug is inevitable

Process 不信任别的 Process，但信任自己全部的 Threads；

User Software 不信任别的 Softwares，但信任 OS；

OS 不信任任何，除了硬件…

从头到尾信任下去，总该有一个「根本」吧？

软件信任的终点直到 BIOS、VMM、kernel loader、kernel……为止。

硬件的信任直到 TPM、secure processor……为止。

Lecture 29 & 30: Isolation

Virtual Machine

虚拟机是虚拟化的基础上的产物。

Why VM?

- 整合性
 - 允许你在一个机器上无缝运行多个 OS
- 隔离性
 - 每个 VM 都互不干扰，错误不扩散
- 可维护性
 - 非常方便部署、备份、迁移
- 安全性
 - 病毒很难逃逸

What are the Differences between OS & VMM?

- Similarities
 - Multiplex hardware
 - Higher privilege
- Differences
 - Different abstraction
 - VMM schedules VMs, OS schedules processes



CPU Virtualization

Virtualization

- CPU virtualization
 - Enable each guest VM has its own kernel and user modes
 - Keep isolation between guest's kernel and user modes
- Memory virtualization
 - Enable each guest VM has its own virtual MMU
 - Keep isolation between guest VMs
- I/O virtualization
 - Enable each guest VM has its own virtual devices

实际上要虚拟化绝对不简单，需要很多硬件共同配合完成。

CPU 要能为每一个 VM 分配各自的 Kernel 和 User Mode；

Memory 要能为每个 VM 分配各自的虚拟 MMU，并保持它们之间的隔离

每一个 VM 都会得到虚拟的 I/O 设备。

CPU Virtualization

CPU Virtualization: Process and VM

- Each process thinks it has the entire CPU
 - Does not care other processes
- OS schedules the processes
 - OS splits the CPU time to time-slices
 - Schedule each process preemptively
 - 3 phases: save context, find next, restore context
- **Idea:** Why not run a VM as a process?

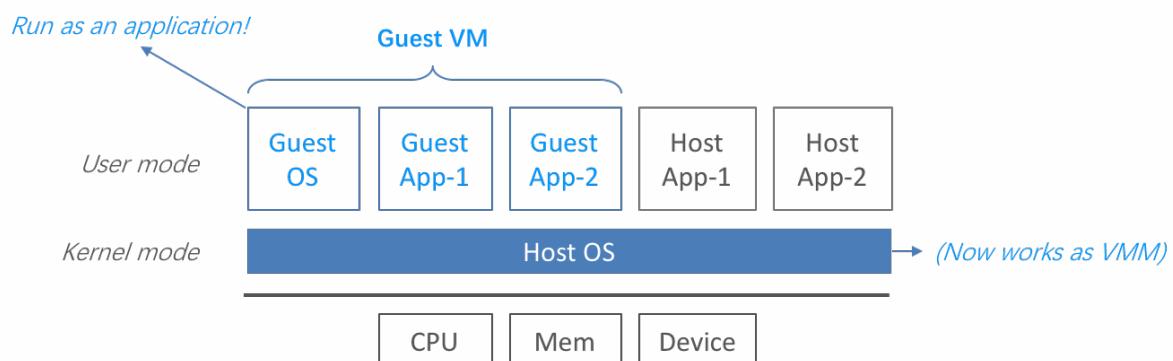
CPU 的虚拟化和 Process 的抽象，有什么异同？

每个 Process 都以为她有全部的 CPU 时间，而不关心其他所有的进程。

这些进程由 OS 负责调度管理，通过分时分片的方式分付各进程使用，并且通过 Process Context 保存与恢复的方式实现了进程的切换。

OS on OS

First Try: OS on OS



我们试着把 OS 作为一个 App 在 Host OS 上运行。也就是，整个 Guest OS 都在用户态运行。

这样做有什么问题？

Run OS as an Application

- Stuck at the very first instruction
 - **cli** : disable interrupt
 - It's a privilege instruction
 - **Cannot run in user mode!**
- Similar instructions
 - E.g., **change CR3, set IDT**, etc.
 - These instructions will change machine states

```
9  
10 .code16  
11 .globl start  
12 start:  
13 cli  
14
```

xv6: bootasm.S

系统启动的时候，会通过一条特殊的指令来禁用中断。

这可是 Privileged Instruction，一般程序不让用的。

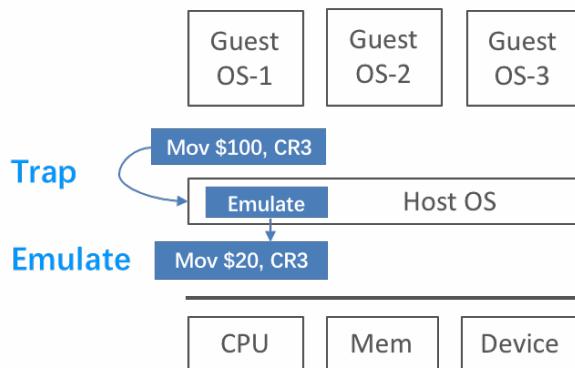
所以这种 OS on OS 的办法，连 OS 都启动不了。

| 呵呵。

Solution: Trap & Emulate

Solution: Trap & Emulate

- **Trap**: running privilege instructions in user-mode will trap to the VMM
- **Emulate**: those instructions are implemented as *functions* in the VMM
 - System states are kept in VMM's memory, and are changed according



既然不能直接调用授权的指令，那我给他包一层吧。

要是 Guest OS 想要执行一条内核模式下才允许的指令，那就陷入 Host OS 的系统中断里，并且由 Host OS 处理之后再 Emulate（模拟），改写指令并发给 CPU。

解决了这个问题，我们的 OS 就能运行在 Application Mode 里了。

Problems: Trap & Emulate

留意到这种解决方法虽然看起来美好，但是并不是所有的体系结构都能够被 Virtualize 的。

有一些结构根本就无法实现这种 Trap & Emulate。

X86 is not Strictly Virtualizable

- Example: the **popf** instruction
 - *popf* takes a word off the stack and puts it into the flags register
 - One flag in that register is the interrupt enable flag (IF)
 - At system level the IF flag is updated by **popf**
 - At user level the IF flag is *not* updated, and CPU **silently** drops updates to the IF
- There **17** such instructions in X86
 - SGDT, SIDT, SLDT, SMSW, PUSHF, POPF, LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT n, RET, STR, MOV

另外，Trap 的时间花费是非常高的。

Fix: Trap & Emulate

对于 x86 那些无法翻译的指令，我们的策略是：

1. Instruction Interpretation
用软件模拟这些指令的结果
2. Binary Translator
把他们转写成其他等价的指令
3. Para-virtualization
把他们在源代码里就给替换掉
4. 换 CPU（大雾）

Instruction Interpretation

Sol-1: Instruction Interpretation

- Emulate **Fetch/Decode/Execute** pipeline in software
 - Emulate all the system status using memory
 - E.g., using an array **GPR [8]** for general purpose registers
 - None guest instruction executes directly on hardware
- E.g., Bochs

- Positives

- Easy to implement & minimal complexity

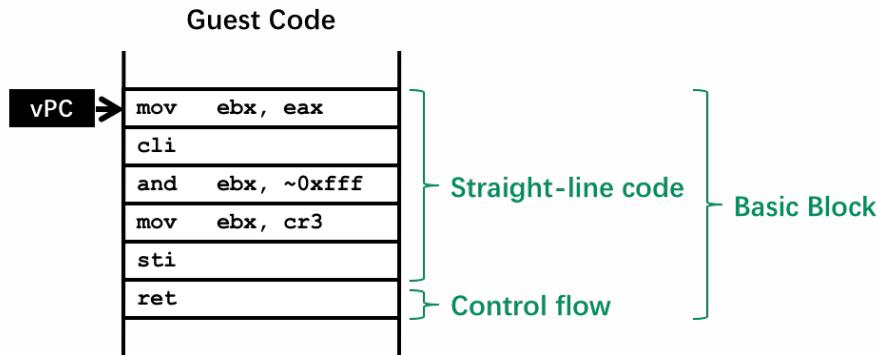
- Negatives

- Very Slow!

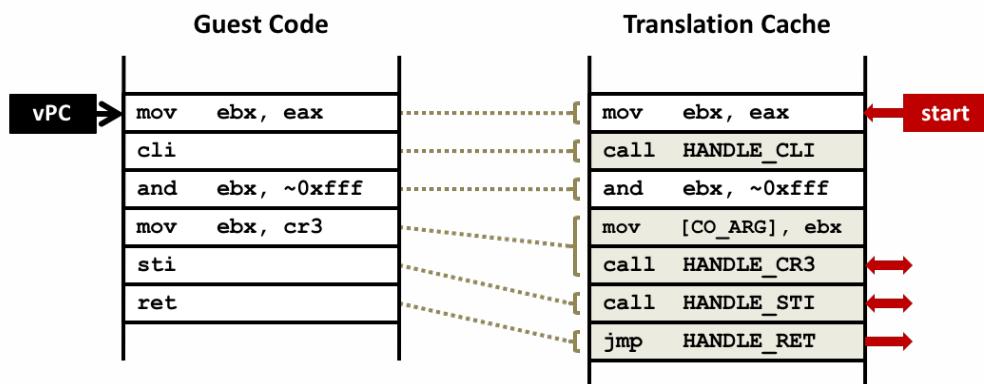
Binary Translator

- Translate before execution
 - Translation unit is basic block (why?)
 - Each basic block -> code cache
 - Translate the 17 instructions to function calls
 - Implemented by the VMM
- E.g., VMware, Qemu

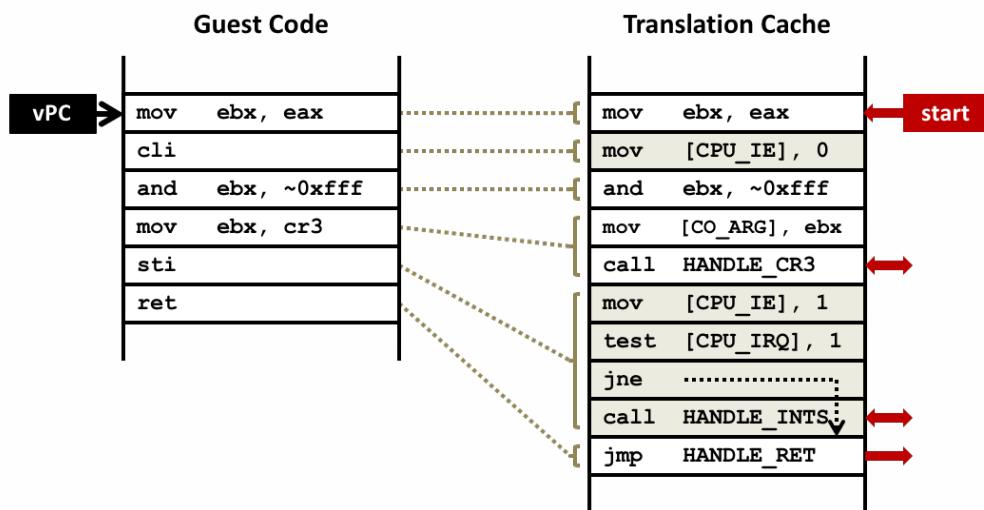
Basic Blocks



Binary Translation



Binary Translation



Issues

- PC synchronization on interrupts
 - Now interrupt will only happen at basic block boundary
 - But on real machine, interrupt may happen at any instruction
- Carefully handle self-modifying code (SMC)

- Notified on writes to translated guest code

Para-virtualization

这就是比较 Dirty 的 Hack 了...

- Modify OS and let it cooperate with the VMM
 - Change sensitive instructions to calls to the VMM
 - Also known as **hypcall**
 - Hypcall can be seen as trap
- E.g., Xen hypervisor
 - Host 100 VMs on 1 machine

Upgrading CPU

这就不了说了...

Memory Virtualization

Virtualizing the Page Tables

虚拟化页表.....

为了区分虚拟化的内存，我们引入一些新的专门名词：

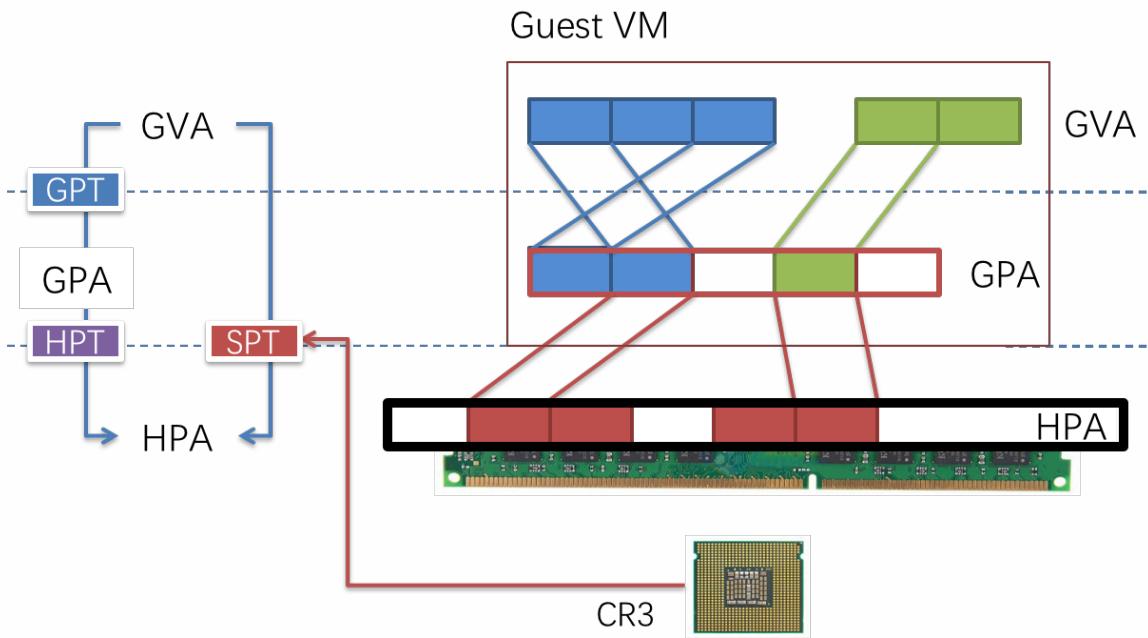
- GVA: Guest Virtual Address
- GPA: Guest Physical Address
- HPA: Host Physical Address

另外要留意，GPA 跟 HPA 不一定一样。一台机器的内存资源不一定全部分配给了虚拟机。

所以直接让 CR3 指向 Guest OS 页表不能用。

Solution 1: Shadow Paging

Solution-1: Shadow Pages



Two Page Tables Become One

1. VMM intercepts guest OS setting the virtual CR3
2. VMM iterates over the guest page table, constructs a corresponding shadow page table
3. In shadow PT, every guest physical address is translated into host physical address
4. Finally, VMM loads the host physical address of the shadow page table

Solution 2: Direct Paging

- Modify the guest OS
 - No GPA is needed, just GVA and HPA
 - Guest OS directly manages its HPA space
 - Use hypercall to let the VMM update the page table
 - The hardware CR3 will point to guest page table
- VMM will check all the page table operations
 - The guest page tables are read-only to the guest

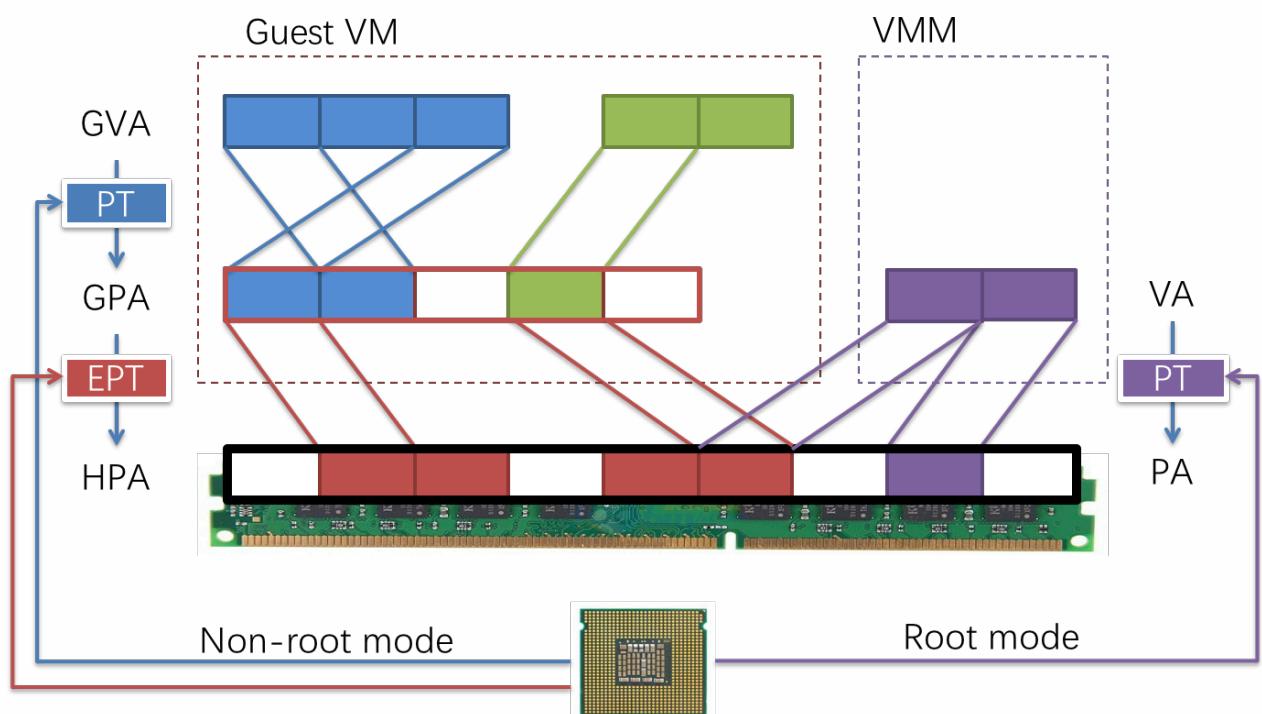
直接不用 GPA，只用 GVA 和 HPA；让 Guest OS 直接管理那一份 HPA 内存。

- Positive
 - Easy to implement and more clear architecture
 - Better performance: guest can batch to reduce trap
- Negatives
 - Not transparent to the guest OS
 - The guest now knows much info, e.g., HPA
 - May use such info to trigger *rowhammer* attacks

Solution 3: Hardware Supported Memory Virtualization

有一些有追求有梦想的硬件提供了内存虚拟化实现：

- Hardware implementation
 - Intel's EPT (Extended Page Table)
 - AMD's NPT (Nested Page Table)
- Another table
 - EPT for translation from **GPA to HPA**
 - EPT is controlled by the hypervisor
 - EPT is per-VM



I/O 虚拟化

OS 再讲。