崩溃一致性

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组

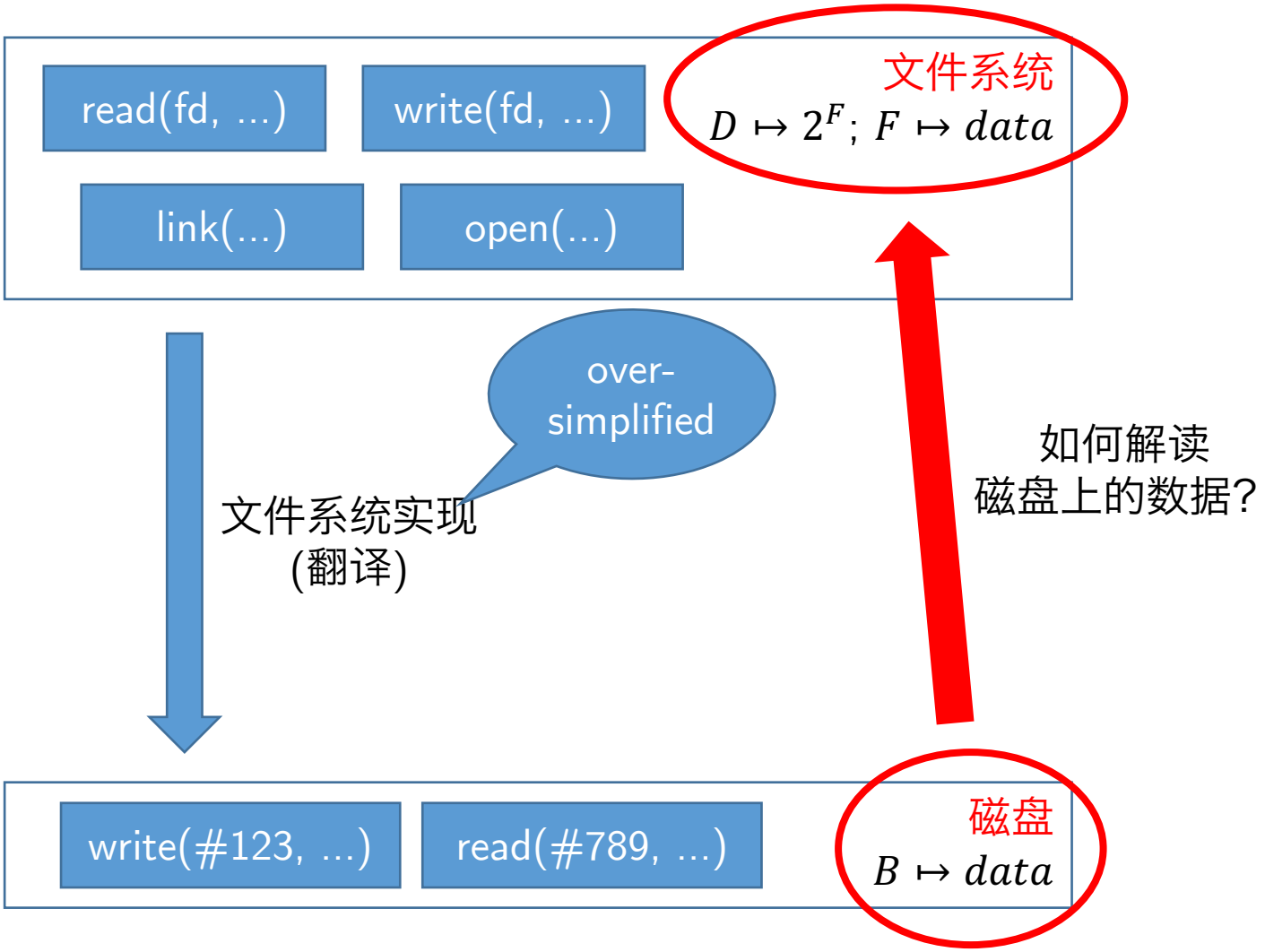
文件系统实现：复习

- 文件由两部分数据组成
  - 文件的元数据(metadata, inode in the UNIX world)
    - 文件名
    - 访问权限 (试图执行不能执行的程序将会返回permission denied)
    - 时间信息 (Make工具用修改时间判断是否需要更新)
    - .....
  - 文件的数据(data)，std::vector<char>
    - 对于普通文件，操作系统不管其中数据的含义
    - 目录也是数据，也可以看成是文件(存储目录里的文件信息)

- 很自然地导出了FAT和ext2的实现
  - so far, so good

问题的提出

回到抽象的角度


read(fd, …)
write(fd, …)
link(…)
open(…)
文件系统
$D \mapsto 2^F$; $F \mapsto data$
over-simplified
文件系统实现
(翻译)
如何解读
磁盘上的数据?
write(#123, …)
read(#789, …)
磁盘
$B \mapsto data$

磁盘：The Difficult Part

- 磁盘
  - 一个简单的数据结构$B \mapsto data$
  - 接收read(#blk, data), write(#blk, data), flush操作序列

- 为了更高效的I/O
  - 磁盘的I/O请求可能被调度
    - write(#x), write(#y)可能被乱序——就像CPU里那样

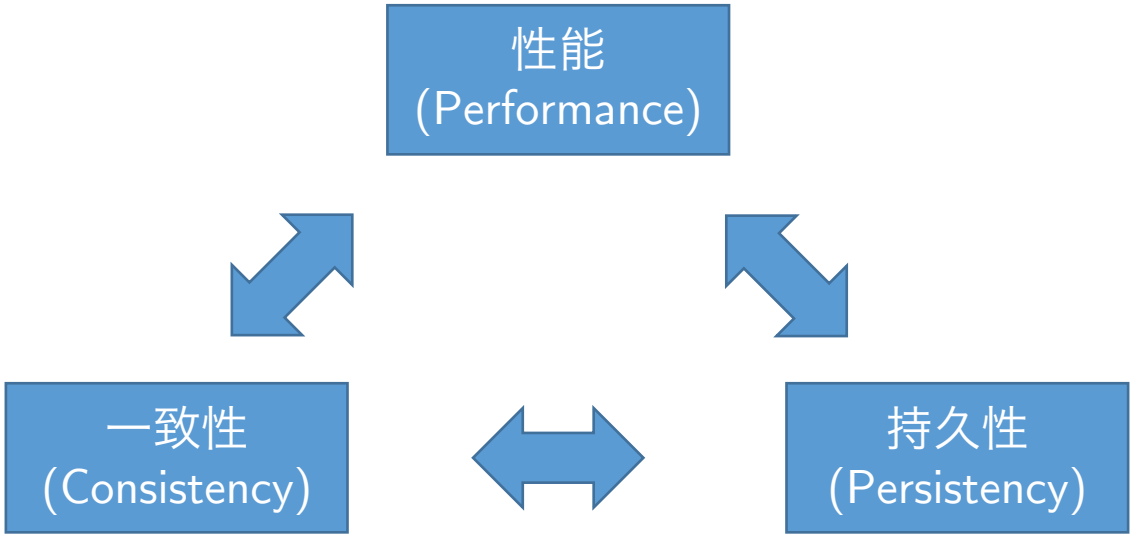
文件系统：The Difficult Part

- 文件系统
  - 一个复杂的数据结构$D \mapsto 2^F$; $F \mapsto data$
  - 支持各类数据结构修改/查询操作(mkdir, unlink, read, ...)

- 为了更好的文件系统(满足应用需求)
  - 性能(performance)：越快越好
  - 一致性(consistency)：磁盘任意时刻的状态能推导出某个过去时刻的文件系统状态
  - 持久性(persistency)：推导出的状态越近越好

Putting Them Together

- 文件系统实现的需求
  - 在不保证顺序写入的磁盘上实现高性能、一致、持久的文件系统
  - 这下就困难了：这些需求是内在冲突的



Some Comments

- 三方面的努力
  - 性能(performance)++：在各级使用缓存掩盖读写延迟
  - 持久性(persistency)++：保证每个写都立即到达磁盘
  - 一致性(consistency)++：？？？
- 本质上说，这些要求是有些矛盾的
  - 计算机系统设计中充满了矛盾，因此需要作出权衡(trade-off)
  - 世界做不到不完美

崩溃一致性

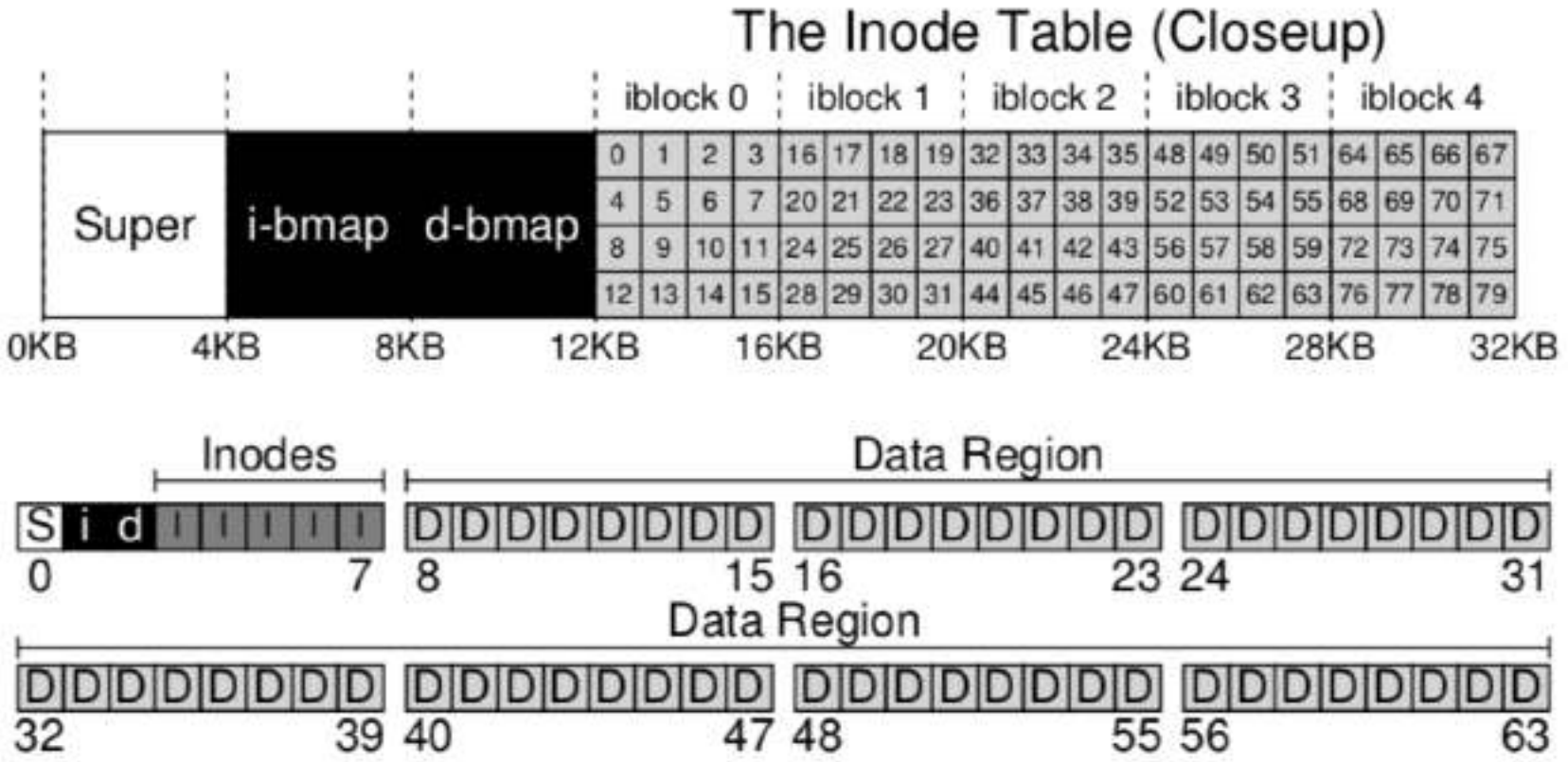
文件系统的崩溃一致性

- “磁盘上的状态能推导出某个过去时刻的文件系统状态”

The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated. After the crash, the system boots and wishes to mount the file system again (in order to access files and such). Given that crashes can occur at arbitrary points in time, how do we ensure the file system keeps the on-disk image in a reasonable state?

回顾：简易UNIX文件系统

- 磁盘上的数据结构



追加写(Appending Write)的崩溃一致性

- 向文件尾部写入数据需要更新若干部分的数据
  - d - 分配一个新的数据块(更新data bitmap)
  - D – 写入实际数据(更新data block)
  - i - 数据块中的数据和索引(更新inode)

- 磁盘上实际会看到{d, D, i}的任意子集
  - 磁盘可能乱序写操作
  - 任何时候都可能崩溃

一些事实

- 如果希望磁盘崩溃时的状态代表某个文件系统的状态，那么

- D → d (数据必须在bitmap之前写入)
  - 否则会读取到不正确的数据
- d → i (bitmap必须在inode之前更新)
  - 否则inode的索引里有#blk，但#blk可能之后分配给其他文件
- i → d (inode必须在bitmap之前更新)
  - 否则分配的数据块就永远没有引用

- 即便是最简单的情况，按照什么顺序写都不对啊……

适当放松一些条件

- 例如允许“dead blocks”
  - D → d → i的写入顺序就是合理的
    - D: 写入一个尚未分配的数据块
    - d: 标记这个数据块已被分配
    - i: 更新文件大小和索引
  - 前提：写操作按顺序到达磁盘(I/O请求不被调度)
    - 但实际是允许调度的，因此在每个写入之后都需要flush

- Dead blocks：垃圾回收
  - 扫描整个文件系统(inodes)，如果有一块没有被任何inode引用则可以释放

实现崩溃一致性：FSCK

File System Checking (FSCK)

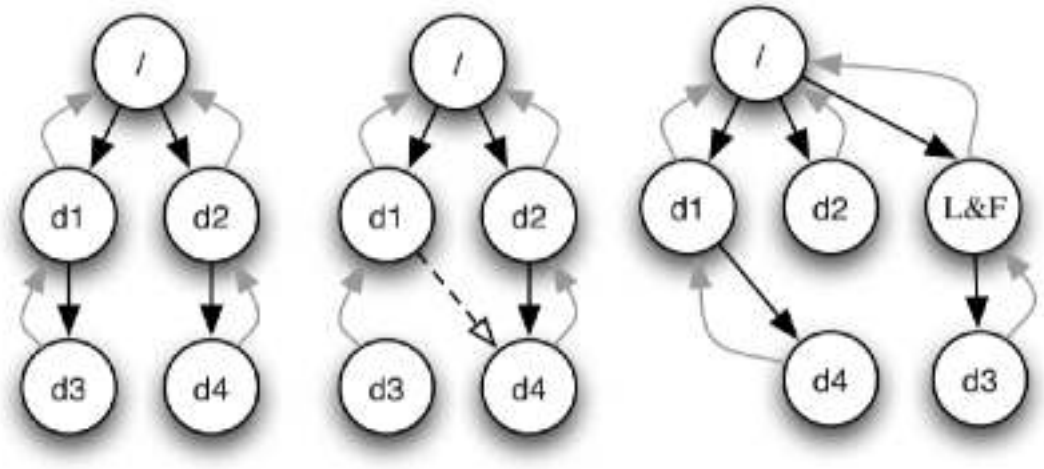
- “不管”崩溃时候的一致性

- 在崩溃以后扫描整个文件系统进行补救
  - 扫描inodes里的所有数据块，检查bitmap的一致性
  - 检查inode数据是否“看起来合法”，否则删除
  - 检查链接情况(没有链接的inode被移到lost+found目录中)
  - 其他完整性的检查……

- 看起来就不靠谱
  - 为了一点小事扫描整个磁盘(Windows也时常这样)
  - 而且没有人能证明这么做一定能回退到一致的文件系统状态

FSCK：的确不靠谱 (需要改进)

- e2fsck (ext2/ext3 fsck)
  - 基于(类似上一页)启发式方法恢复
  - 既可能损坏文件系统，也可能恢复的不对



H S Gunawi, A Rajimwale, A C Arpaci-Dusseau, R H Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proc. of OSDI*, 2008.

实现崩溃一致性：事务

实现崩溃一致性：另一个角度

- “Recovery code is complex and hard to get right”
  - 从另一个角度考虑这个问题
  - 我们只是希望写三个磁盘数据块{D, d, i}

- 希望有一个机制为我们提供原子性
  - “上锁”
  - “写入”
  - “解锁前不允许崩溃”
  - 这问题就解决了

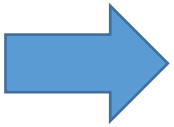
```
void update() {
  lock(&lk);
  write(#D);
  write(#d);
  write(#i); // 无所谓顺序
  unlock(&lk);
}
```

文件系统状态的两种表示

- 文件系统 = 数据结构
  - 一个复杂的数据结构$D \mapsto 2^F$; $F \mapsto data$
  - 支持查询(read, readdir, ...)和修改(write, unlink, ...)操作

- 一方面，我们用目录树表示文件系统
- 另一方面，所有历史上的修改操作唯一确定了一个目录树

mkdir(...)
+ mkdir(...)
+ link(...)
+ ...



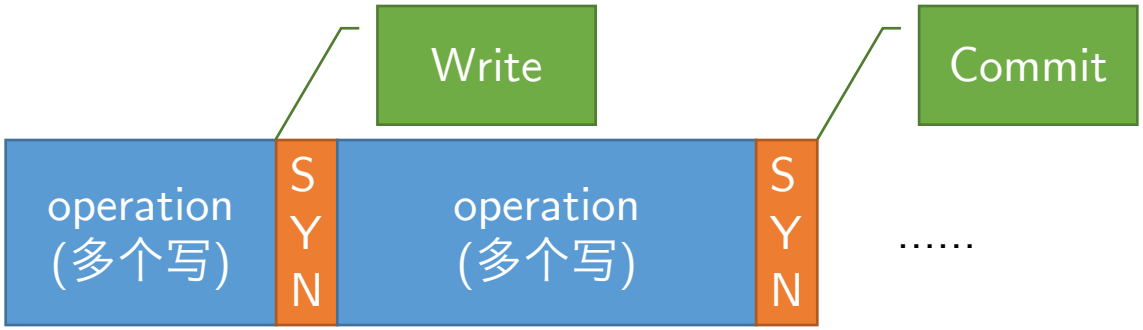
一个疯狂的想法

- 我们不在磁盘上维护任何数据结构，而只是记录下所有历史文件系统的更新操作
- 每次挂载文件系统后，都在内存中重建文件系统

- 实现
  - append-only write
  - 问题：这样能保证崩溃一致性吗?

疯狂的想法：实现崩溃一致性

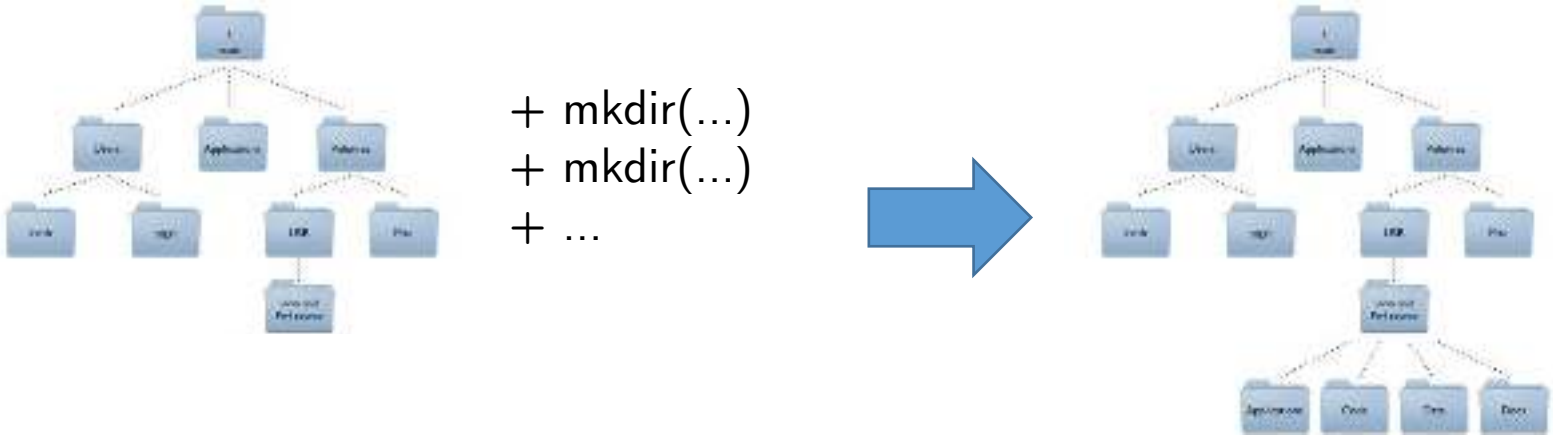
- 先写操作(可以有多个writes)，等待写入磁盘
- 然后再写入SYN (看到SYN → 操作已经写入)



- 崩溃恢复
  - 不断读operations，直到第一个没有SYN的停止
  - 依靠SYN保证了写入的原子性
    - 每个operation是一个事务事务(Transaction)
  - It works! (M6 libkvdb可以使用这个方法)

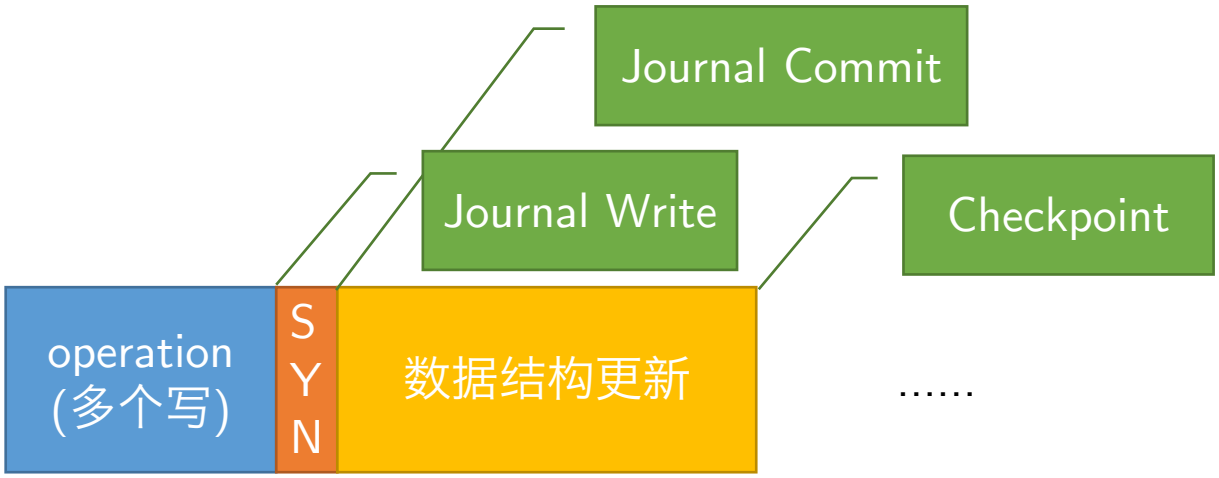
疯狂的想法：改进

- 在内存里replay所有操作太浪费了
  - 我们只要从某个一致的文件系统出发，执行之后的修改操作，就能得到一个一致的文件系统



Journaling File System

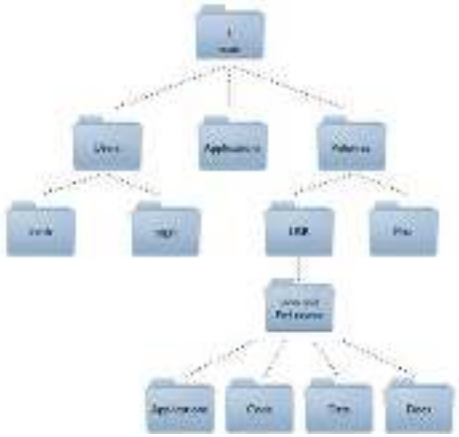
- 既维护操作序列，也维护数据结构
  - 如果执行操作，首先做记录(journaling)
  - 然后更新磁盘上的数据结构
  - .....



Journaling File System (cont'd)

- 崩溃发生时，磁盘上的数据结构可能是不一致的
  - 但我们只要replay journal (操作)就能保证到达一致的状态
  - 已经完成的操作不需要replay (由journal管理)

checkpoint



新的checkpoint

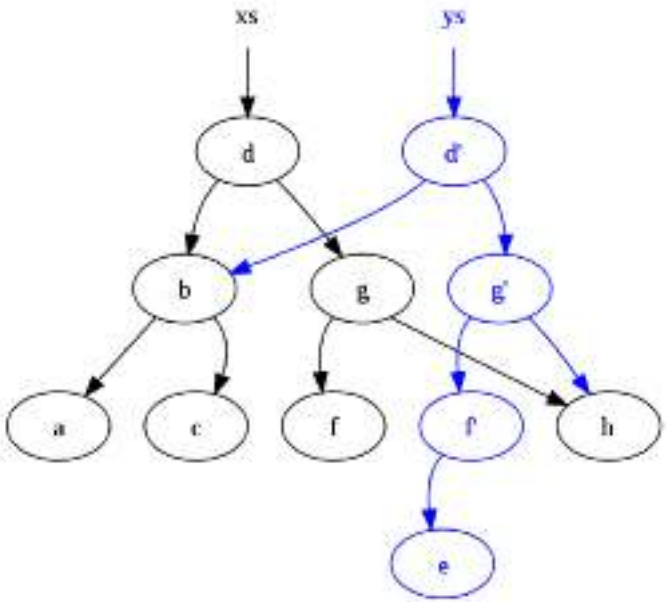
优化

- Meta-data journaling
  - 先写数据，这样就不用journal data了 (几乎减半)
  - tricky
- Batching
  - 可以累积多个操作后一起写入(系统中的jbd)
- Journal的数据结构维护
  - Journal是个vector<TX>，操作:
    - push_back -- 新事务
    - pop_back -- 已经写入磁盘的事务无需维护

实现崩溃一致性：写时复制

一个(也许更)疯狂的想法

Ordinary data structures are *ephemeral* (ext2/3, ...) in the sense that making a change to the structure destroys the old version, leaving only the new one... We shall call a data structure *persistent* if it supports access to multiple versions.



d(b(a, c), g(f, h))

在f的左孩子插入e

d'(b(a, c), g'(f'(e'), h))

A Little-bit Theory

- 神奇的结论

> Any pointer-machine data structure with no more than $O(1)$ pointers to any nodes (in any version) can be made partially persistent with $O(1)$ amortized multiplicative overhead and $O(1)$ space per change.
>
> --- Driscoll, Sarnak, Sleator, Tarjan - JCSS 1989

- 以及很多结构都能实现pure functional
  - 也就是只需要append-only写操作就能维护
  - 与生俱来的崩溃一致性
    - 好像保存了所有操作的序列
    - 但又同时维护了数据结构(!)

回到文件系统实现

文件系统实现

- 在性能、持久性、一致性之间作出权衡
  - 使用缓存保持高性能
  - 使用日志实现一致性
  - 牺牲立即持久性